MASTER EN SCIENCES INFORMATIQUES

Conception et réalisation d'une boite à outils pour un environnement de spécification

El Ayeb, Bechir

Award date:
1984

Awarding institution:
Universite de Namur

FACULTES UNIVERSITAIRES
NAMUR (BELGIQUE)
Institut d'informatique

Année académiqúe 1983 - 1984

CONCEPTION ET REALISATION D'UNE BOITE A OUTILS POUR UN ENVIRONNEMENT DE SPECIFICATION

Mémoire présenté par Bechir EL-AYEB
en vue de l'obtention
du grade de licencié et maître
en informatique

REMERCIEMENTS

Avant d'aborder le corps de ce travail, nous tenons à remercier toutes les personnes qui, à titre divers, lui ont permis de voir le jour.

Nos remerciements vont tout d'abord à Monsieur Axel LAMSWEERDE, professeur à l'université de Namur et promoteur de ce travail. Sa compétence scientifique, son attention, ses conseils furent d'une valeur sans égale.

Nous tenons à remercier les membres du Centre de Recherches en Informatique de Nancy pour l'accueil qu'ils nous ont réservé. Parmi eux, nous remercions plus particulièrement :

. Messieurs Jean-Pierre FINANCE et Alain QUERE, professeurs à l'université de Nancy, pour leur acceuil, leurs critiques pertinentes émises sur ce travail;

. Monsieur Jean-Pierre JACQUOT qui nous fut d'une aide précieuse pour l'apprentissage du logiciel CEYX et avec qui, nous avons eu, à tout moment, des discussions fructueuses;

. Madame Annie Ressouche qui nous a facilité la compréhension du logiciel LEYACC et qui nous a aidé dans le développement de l'analyseur lexical.

Tous nos remerciements vont également à Monsieur François GEORGES pour la présentation et la réalisation dactylographique de ce travail.

TABLES DE MATIERES

"L'évidence est toujours l'ennemie de la correction. C'est pourquoi on invente un symbolisme nouveau et difficile dans lequel rien ne semble évident. Nous définissons ensuite les règles d'emploi de ces symboles et tout devient mécanique."

Bertrand RUSSEL,
Les Mathématiques et les Métaphysiciens

INTRODUCTION

Ecrire une bonne spécification d'un problème à résoudre par ordinateur est un des points épineux de l'informatique.
La spécification est une étape critique car elle a pour objet de convertir une formulation imprécise du problème à résoudre en une formulation précise. Les ouvrages traitant ce sujet sont de plus en plus abondants. On peut citer les références [LAM;82], [GER;80], [DUB;82], [FIN;73].

On peut retenir deux types d'approches développés pour analyser et spécifier un problème [FIN;83].

. Les approches dites fonctionnelles (SREM [SREN;80], réseaux de PETRI [215;78], HIPO [IBM;75], ...) mettant l'accent sur la description des traitements (ou fonctions) que doit réaliser le système informatique étudié. Un inconvénient majeur de ce point de vue est la faible évolutivité des systèmes obtenus, due en partie à la faiblesse de la structuration des données. On pourra remarquer qu'une faible évolutivité peut être causée par un fort couplage de traitements, celui-ci pouvant être dû à une faible structuration de données. Notons encore un autre inconvénient de cette approche : Les fortes redondances possibles dans les données. Ceci est dû en partie à l'effort accordé plus aux traitements qu'aux données.

. D'autres approches s'orientent davantage vers la description des objets sur lesquels le système informatique va opérer et de leurs relations (ISDOS [TEI;79], MERISE [TAR;79], Z [DEM;80], ...). De telles approches reposent sur des modèles tels que le modèle Entité / Association [BOD;84], le modèle relationnel [HAINN], les types abstraits [LIS;74].

On peut écrire les spécifications dans un langage formel de spécification ou en langage naturel. Les langages formels reflètent généralement un des types d'approche esquissés ci-dessus. L'utilisation d'un langage formel de spécification présente plusieurs avantages. On peut citer :

. la possibilté d'automatiser un certain nombre de vérifications (cohérence, relative complétude, etc);

. la possibilité d'éliminer les ambiguïtés;

. la perspective de pouvoir dériver automatiquement ou semi-automatiquement d'une maquette "exécutable" à partir d'une spécification formelle.

A l'état actuel des recherches en informatique on est encore loin de concevoir des outils permettant la dérivation automatique des programmes couvrant tous les domaines. On se tourne alors vers la conception assistée par ordinateur (CAO). Celle-ci est un ensemble d'outils logiciels aidant à la spécification, à la conception et à la mise au point des programmes.

Voici quelques fonctionnalités typiques de tels outils. Le lecteur intéressé peut trouver une étude rigoureuse présentée par Axel Van Lamsweerde [LAM;82].

En toute généralité, on peut distinguer d'abord deux types d'outils quant à leur nature :

. les outils actifs assistant directement dans l'application d'une méthodologie précise;

. les outils passifs réduisant fortement les aspects non créatifs et fastidieux du travail.

Citons maintenant les fonctionnalités typiques par cycle de vie :

. Outils d'aide à la spécification. Cette catégorie d'outils est très récente. La principale préoccupation à ce stade est l'organisation et la documentation des spécifications.
On peut alors noter par exemple des outils (ou fonctions) de mise à jour, de contrôle de cohérence, de détection d'erreurs, de nettoyage, de simulation et de formattage des spécifications. Certains systèmes se dotent d'un (de) langage(s) formaté(s) de spécifications, d'un (de) modèle(s) pour "structurer" le(s) traitement(s) et/ou la (les) donnée(s). Parmi ces outils, on peut citer :

- le système ISDOS (Techroew et Hershey (1977) [TEI 79];
- les outils d'aide SA [DEM;78], [DEL 82];
- les outils d'aide à la spécification dans l'environnement USE [WAS;79];
- l'environnement ZAIDE [DEM;80].

. Outils d'aide à la conception. On relèvera ici deux aspects : d'une part la conception d'une architecture logicielle et d'autre part la construction de différents programmes au sein de cette architecture. Le problème de conception reste complexe et peu compris. Voici quelques références : [SCH;81], [HAM;76], [KER;81]. Pour une étude et une bibliographie plus complètes, le lecteur pourra se référer à [LAM;82].

. Outils d'aide à la mise au point. A cette étape de cycle de vie, les outils sont les plus nombreux. On relève ainsi la mise en place de langages mieux adaptés à un type particulier d'application, tels que le langage Plain [WAS;81], des éditeurs et formatteurs guidés par la syntaxe [DON;79], [MAS;81], [RIN;81] et aussi des préprocesseurs et générateurs de codes dans un langage standardisé (Cobol, Fortran) à partir de prgrammes rédigés dans un langage de haut niveau [WEI;76], [WEL;86].

C'est dans le contexte d'outils d'aide à la spécification que s'est inséré notre travail.

Le projet SPES (développé au chapitre 1) vise à constituer un environnement de spécification autour du langage SPES et de la méthode déductive (tous deux développés au chapitre 1).

L'objectif de notre travail est de fournir une boîte à outils afin de permettre la réalisation du projet SPES.

Les grandes lignes de réalisation de notre travail furent les suivantes :

- pour des raisons expliquées plus loin, la structure de données retenue est l'arbre;
- développer un ensemble de primitives en Lisp permettant la gestion d'une forêt d'arbres (création, suppression, insertion, visualisation);

- développer d'autres primitives en Lisp d'exploration de la forêt d'arbres. Cette exploration a pour but de fournir un diagnostic (résultat d'analyse d'un ou plusieurs énoncés) sur la forêt d'arbres;
- générer un analyseur syntaxique pour le langage SPES;
- concevoir et réaliser un analyseur lexical;
- mettre en oeuvre l'ensemble des primitives citées afin de montrer la possibilité d'exploitation des outils fournis.

Le présent mémoire comprend six chapitres.

Le premier chapitre précisera le cadre de notre travail. Le projet SPES, la méthode déductive et la langage SPES seront présentés brièvement. Une présentation de l'environnement de base disponible cloture le chapitre.

Le deuxième chapitre consiste en une première présentation de la boîte à outils. Tout d'abord, un bref aperçu sur l'édition syntaxique; ensuite, une ébauche d'explication sur l'architecture globale de la boîte à outils ainsi que la philosophie architecturale.

Le troisième chapitre présentera de façon plus précise le contenu de la boîte à outils.

Le quatrième chapitre traitera l'analyseur syntaxique et l'analyseur lexical.

Le cinquième chapitre donnera un aperçu des extensions possibles de la boîte à outils conçue et réalisée.

Nous consacrerons le sixième chapitre à nos conclusions.

"if you want more effective
programmers, you will discover
that they should not waste their
time debugging, they should not
introduce the bugs to start with"

DIJKSTRA

CHAPITRE 1 LE CADRE DE NOTRE TRAVAIL

Ce chapitre présentera tout d'abord le projet SPES, la méthode déductive, le langage SPES et ensuite l'environnement de base.

1.1. LE PROJET SPES [FIN; 83]

1.1.1. Objectifs

Le projet SPES vise la construction et la transformation de façon méthodique d'une spécification formelle d'un problème informatique.
La construction vise à :
- exprimer les relations entre résultats et données dans des modules appelés énoncés;
- mettre la structure de l'univers (ou "système d'information") sous forme de description de types abstraits de données.

L'étape de transformation vise alors à obtenir un programme "exécutable", de façon systématique à partir de la spécification ainsi construite.

Le projet SPES a d'autres préoccupations, par exemple la dérivation d'un texte en langage naturel à partir d'une spécification formelle ou la déduction d'une spécification formelle à partir d'une spécification en langage naturel.

1.1.2. La méthode déductive

La démarche que nous allons exposer très brièvement présente une réflexion menée à Nancy depuis 1973 dans le cadre d'une méthodologie de la programmation. Les détails peuvent être trouvés dans [PAI; 79] .

Les idées de base sont les suivantes :

. Commencer à caractériser le résultat associé au problème considéré en fonction des "résultats";

. Ces "résultats" intermédiaires définissent de nouveaux problèmes auxiliaires auxquels on va récursivement appliquer la première et la deuxieme idées mentionnées;

. On arrête le processus quand tous les résultats intermédiaires sont les données du problème de départ.

L'exemple extrêmement simple qui suit illustre la méthode.

Soit à calculer le montant d'une facture à payer.

M : montant de la facture à payer	$M = S - R$
S : montant total de la facture	$S = \sum_{i=1}^{L} P_i Q_i$
R : remise accordée	R = Pourcent * S
L : nombre de lignes de la facture	L = donnée
Q_i : quantité figurant sur la ligne i	Q_i = donnée
P_i : prix unitaire figurant sur la ligne i	P_i = donnée
Pourcent : pourcentage accordé	Pourcent = donnée

Partant de la définition du montant à payer nous avons introduit des résultats intermédiaires S et R qui ont à leur tour subi la même démarche. On s'est arrêté lorsque chaque "résultat" est une donnée du problème de départ.

Un problème majeur en programmation est la multitude de questions que l'on peut se poser en même temps. On remarquera ici que la méthode guide les questions à se poser.
On peut aussi remarquer que la résolution de ce problème extrêmement simple peut être schématisée en un arbre. La racine constituant le calcul de M s'est ramifiée en deux sous-problèmes (ou deux branches) : le calcul de S et le calcul de R . . .
Cette ramification peut guider les questions à se poser.

1.1.3. La démarche SPES

Au début de l'introduction nous avons cité deux types d'approche pour spécifier un problème. La démarche SPES constitue un compromis entre ces deux points de vue.
Au départ, le spécifieur caractérise la relation associant données et résultats dans le problème considéré (ou éventuellement la famille de plusieurs relations, si le problème doit

réaliser plusieurs fonctions). La spécifications précise de cette relation conduit à introduire progressivement des objets et des fonctions intermédiaires qui doivent à leur tour être spécifiées. Ce processus récursif s'effectue par spécialisation successives de fonctions et d'objets, guidés par une analyse de résultats selon une stratégie déductive tel que celle exposé en 1.1.2. Le lecteur intéressé par plus de détails peut se référer à [FIN;83], [FIN;79a] [DUB;78].

Les objets introduits progressivement sont typés en sortes qui se structurent en une famille de types abstraits hiérarchiques. La hiérarchie ainsi obtenue peut être arbitraire et contraignante si l'on veut faire évoluer le système, soit par ajout de nouvelles fonctionnalités soit pour les transformer en système "exécutable". Pour résoudre cette difficulté, des outils SPES sont à la disposition du spécifieur pour réorganiser une structure hiérarchique de types afin de représenter les transformations souhaitées.

1.1.4. Le langage SPES

Ce langage formel de spécification permet d'exprimer simplement et de façon PRECISE un problème informatique sans se préoccuper de sa résolution en encourageant une démarche déductive. Quelques exemples nous permettront de mettre en évidence les particularités de ce langage fondé sur la logique de premier ordre et la notion des types abstraits de données pour lesquels les fondements sont bien établis. Un exposé plus détaillé incluant la grammaire du langage pourra être trouvée dans [QUE;83].

Exemple 1

```
(1)  PGCD : (x ; a,b)
(2)  x ? entier pgcd de 2 entiers a et b
(3)  x : ENTIER
(4)  x tq div(x,a) et div(x,b) et qq y:ENTIER
(5)       ((div(y,a) et div(y,b)= div(y,x)))
(6)  a = donnée
(7)  a : ENTIER
(8)  a,b ? entiers dont on veut calculer le pgcd
(9)  b : ENTIER
(10) b = donnée
```

L'énoncé ci dessus définit le pgcd de deux entiers a et b.
Les caractères gras sont les terminaux du langage.

Commentons successivement les lignes (1) à (10).

(1) Est l'en-tête de l'énoncé. Il est formé de l'identificateur de l'énoncé (en majuscule) et d'une liste entre parenthèses. Cette liste est subdivisée en deux sous-listes séparées par un point-virgule. La première sous-liste représente l' (les) identificateur(s) de résultat(s). La deuxième sous-liste représente l' (les) identificacteur(s) de donnée(s). En absence de donnée(s) cette dernière sous-liste peut être omise.

(2) Cette ligne constitue une définition. Celle-ci, de même que que celles des lignes (3), (4) et (5) comprennent en partie gauche le(s) objet(s) à définir. Cette partie gauche est suivie d'un symbole ou d'un mot réservé qui précise la forme de la définition. En partie droite se trouve le corps de la définition. Ici, "?" introduit une définition informelle, texte qui peut apparaître comme un simple commentaire.

(3) Le caractère ":" introduit une définition de sorte (ou type); ici, la sorte des entiers naturels notée ENTIER.

(4) et (5) Le mot réservé "tq" introduit une définition formelle de l'objet x par un prédicat dépendant de x et éventuellement d'autre(s) objet(s) (ici, les objets a,b). Le corps de la définition apparaît comme une expression à résultat booléen. Intuitivement, cette définition signifie : a,b étant données, trouver x de façon que le corps de la définition soit vrai. L'identificateur div représente un opérateur associant à deux entiers un booléen. On suppose ici que cet opérateur est défini par ailleurs : div(a,b) est vrai si le reste de la division de a par b est nul. Le mot réservé "qq" est le qualificateur universel; il est suivi d'un identificateur y (défini de façon locale, la portée de y est limitée à cette définition) et dont la sorte est ENTIER. Le symbole "=>" désigne l'implication. Cette définition de l'objet x est dite "implicite" (à l'opposé de "explicite" : voir commentaire de la ligne suivante et celui de la 9e ligne de l'exemple 2).

(6) et (10) C'est la définition formelle des objets a et b. Cette définition est dite explicite (se prêtant davantage à une traduction ou "tronsformation" en langage de programmation ce qui est une étape ultérieure du projet SPES).
(7) et (9) C'est la définition de la sorte de a et b.
(8) C'est la définition informelle de a et b.

Exemple 2

```
(1) PPCM(r;u,v)
(2) r ? ppcm de u et v
(3) r : ENTIER
(4) u,v ? entier données
(5) u,v = donnée
(6) u,v : ENTIER
(7) r tq r*y=u*v
(8) y ? pgcd de u et v
(9) y tq PGCD (y;u,v)
(10) y : ENTIER
```

Seule la ligne (3) introduit un nouveau concept. C'est celui de la définition d'un objet par un appel (une référence) à un autre énoncé. Ici on définit y en se référant à l'énoncé PGCD.

Exemple 3

```
(1) PGCD(x;a,b)
     .
     .
     .
(4) x=jqa v=0 rep w=u u=v v=reste(w,v) init u=a
(5) v=b
     .
     .
     .
```

Cet énoncé correspond au même énoncé que celui de l'exemple 1. Seules les lignes 4 et 5 ont changé. A l'exemple 1 l'objet (résultat) x a été défini par un prédicat. Ici, il a été défini de façon "algorithmique", au moyen d'une boucle commençant par le mot réservé "jqa". Le corps de la répétitive est précédé du mot réservé "rep". Le corps se termine par le mot réservé "init" (initialisation) où on a défini les objets u,v. Cette définition de l'objet x est dite explicite. Le passage automatique ou semi-

automatique des lignes (4) et (5) de l'exemple 1 aux mêmes lignes de l'exemple 3 constitue la transformation. On peut noter que la dérivation d'un programme "exécutable" de l'exemple 3 est facile.

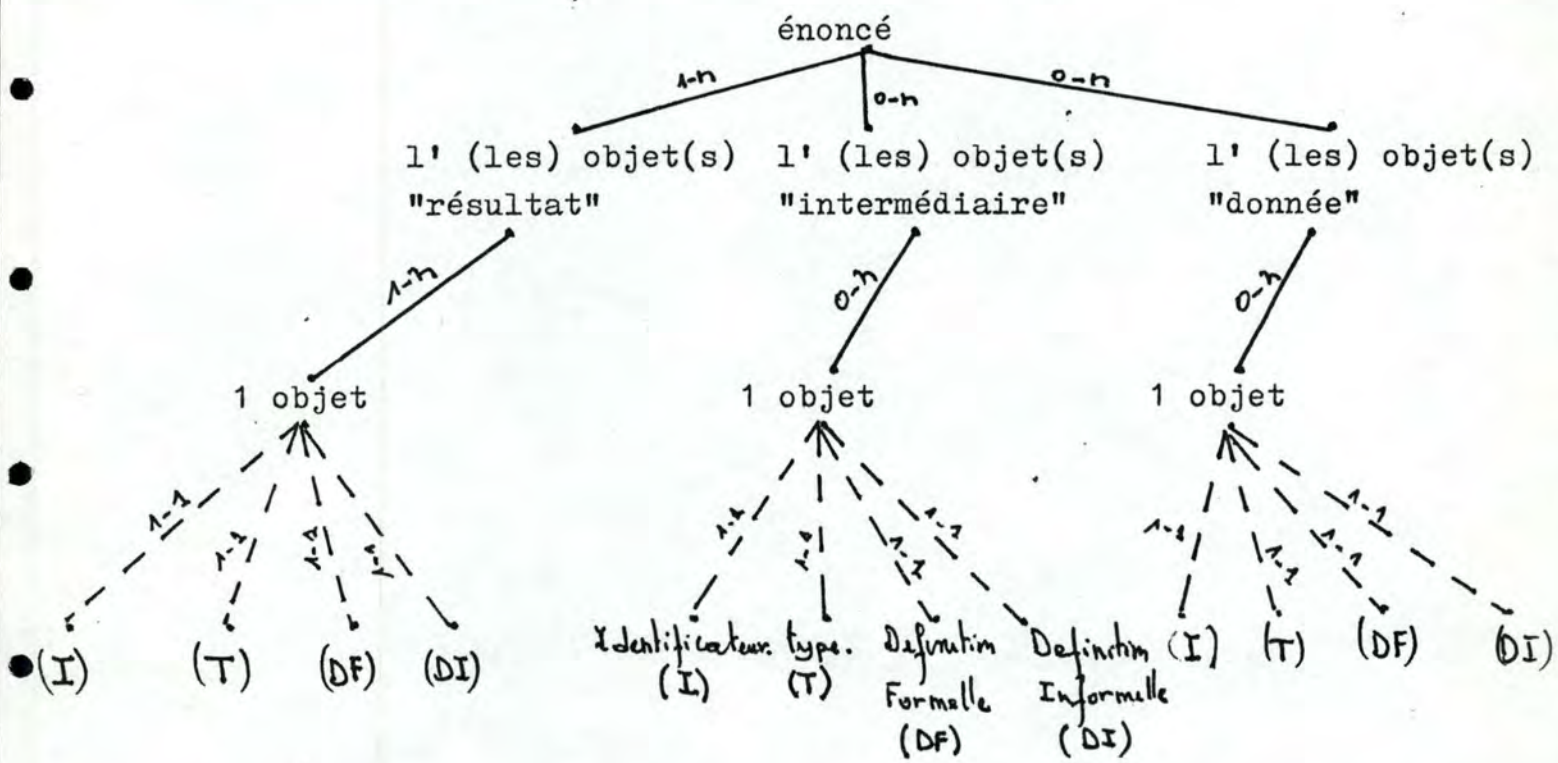
Ces exemples illustrent quelques particularités caractéristiques du langage SPES :

(a) Chaque énoncé contient trois catégories d'objets
- . les objets "résultat" (cf. r dans l'exemple 2);
- . les objets "donnée" (cf. u et v dans l'exemple 2)
  Rappelons que cette catégorie n'est pas obligatoire.
  Un énoncé peut ne pas avoir de données;
- . les autres objets (cf. y dans l'exemple 2),
  appelés objets "intermédiaire".

(b) Chaque objet possède
- . un identificateur;
- . une sorte;
- . une définition formelle;
- . une définition informelle

à l'exception des objets locaux (cf. y dans l'exemple 1).

(c) D'après (a) et (b) on peut dresser le schéma suivant représentant la structure d'un énoncé :



Un arc en trait continu signifie "se compose de" : un énoncé est composé des objets "résultat", "intermédiaire" et "donnée".
Un arc en trait discontinu signifie "est défini par".
La connectivité est indiquée sur chaque arc.

(d) Seule la sorte ENTIER a été utilisée dans les exemples ci-dessus. Cette sorte est dite prédéfinie; on pourrait introduire d'autres sortes, non prédéfinies, pour définir les objets. Ces sortes non prédéfinies devraient être définies dans une bibliothèque de sortes. Cette définition de sorte (non prédéfinie) est une description en termes de types abstraits de données [FIN;79]. L'ensemble de telles descriptions constitue ce qu'on appelle la bibliothèque des sortes pour le problème considéré. Toute description de type doit répondre à une syntaxe que l'on trouve dans [QUE;83].

1.1.5. Les outils SPES

Il paraît souhaitable que le spécifieur puisse se concentrer sur les décisions successives qu'il doit prendre ou remettre en cause plutôt que sur les manipulations fastidieuses des textes d'énoncés. La conception souple d'une spécification ainsi que son évolution nécessitent donc l'utilisation d'outils logiciels élaborés réalisant des fonctions (l'édition, la visualisation, la transformation, l'archivage) permettant une forte structuration distinguant ainsi les aspects fondamentaux des aspects secondaires. L'utilité du système SPES résultera également de sa capacité conduisant d'un énoncé initial d'un problème aux divers sous-énoncés puis à diverses solutions algorithmiques. Notons que les différentes branches d'un tel arbre seront étiquetés par des opérateurs permettant d'enrichir une spécification ou de transformer un énoncé.

Des outils au projet SPES ont été développés par Michel Buyse et Paul Van Hemelryck [BMP;83]. Ces outils utilisent des logiciels tels que MENTOR, SYNTAX et sont écrits en Pascal. Parmi ces

outils on peut citer l'édition syntaxique, la décompilation, les analyses de cohérence des textes d'énoncés, la production des récapitulatifs. Cette expérience révèle un manque de souplesse dû aux logiciels. Ce manque de souplesse est dû en partie au caractère "boîte noire" des logiciels utilisés. Se rendant compte de ce manque de souplesse, Jean-Pierre Jacquot a développé une mini-expérience de certains outils (l'éditeur syntaxique) sous le logiciel CEYX. Cette mini-expérience paraît encourageante.

1.2. L'ENVIRONNMENT DE BASE DISPONIBLE

Rappelons que le projet SPES vise la construction et la transformation (en vue de l'obtention des programmes exécutables à partir d'un énoncé) des spécifications (ou énoncés).
Rappelons aussi que l'objectif de notre mémoire est de fournir des primitives pour la gestion d'une forêt d'arbres. Ces primitives seront la base pour la mise en place de l'éditeur syntaxique (construction des énoncés). D'autres primitives développées serviront à la visualisation, à l'analyse ... des énoncés.

1.2.1. Lisp

Ce langage est celui qui a été retenu pour développer le projet SPES. Rappelons qu'une première version a été développée en Pascal compatible avec les logiciels de base et une rigidité est apparue (cf. 1.1.5, p.15) celle-ci étant due au caractère des logiciels de base. Le logiciel CEYX utilisé à la mini-expérience est compatible avec Lisp. C'est pour cette raison que Lisp a été retenu.

La structure de données en Lisp est fondée sur les listes et les atomes. Pour pouvoir manipuler une structure de données tel que un arbre, une table, une file, il suffit de la simuler en listes et de se doter des fonctions (ou primitives) permettant de manipuler sa structure de données. Notons que la simulation d'une structure de données et la création de fonctions appropriées est fort souple en Lisp vu la puissance de ce langage pour manipuler les listes. De cette façon, on peut "se tailler" un Lisp à la mesure de ses propres besoins. Cette souplesse n'est cependant pas qu'un avantage : on assiste en effet à une multiplication des versions (Vlisp, Bvlisp, Lelisp, Franzlisp, ...), chacune étant enrichie à la guise de son auteur. A première vue, ces différents versions pourront porter atteinte à la portabilité des logiciels écrits en Lisp. Heureusement la souplesse de Lisp peut remédier à cet inconvénient : il suffit d'enrichir son Lisp afin de répondre aux exigences du logiciel utilisé. On entend par enrichissement la création de nouvelles fonctions sans modifier le logiciel utilisé. De manière alternative, il nous

semblerait naturel de changer seulement ce qui englobe (l'environnement disponible) le logiciel utilisé. En effet, si nous considérons le logiciel comme un "macro-programme", celui-ci possède des pré-conditions (l'existence de fonctions de base, ...). Il suffirait ainsi de lui assurer de telles pré-conditions au lieu de modifier le logiciel utilisé pour tenir compte des nouvelles pré-conditions relatives au nouvel environnment. La fonctionnalité de Lisp permet facilement (par ajout de fonctions) d'assurer les nouvelles conditions. Malheureusement, dans de nombreux cas où les logiciels sont écrits dans des langages non fonctionnels on ne sait pas appliquer cette démarche.

1.2.2. CEYX

La version Lelisp développée par Jérôme Chailloux à l'INRIA qui était à notre disposition n'offre pas de structure d'arborescence en tant que telle. Par contre le logiciel CEYX écrit en Lelisp et développé à l'INRIA par Jean-Marie Hullot permet le traitement des arbres typés.

Rappelons que les raisons du choix de la structure de données seront données dans le chapitre 2 (p.26). Il était alors intéressant de disposer des facilités permettant la gestion des arbres. Ce logiciel expérimental CEYX n'offre pas une très grande facilité. Par contre on peut l'enrichir par création de nouvelles fonctions.

Donnons quelques définitions avant d'examiner certaines possibilités de CEYX.

DEFINITIONS

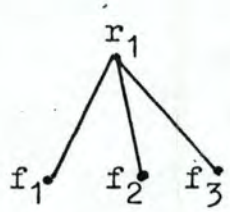


fig. 1

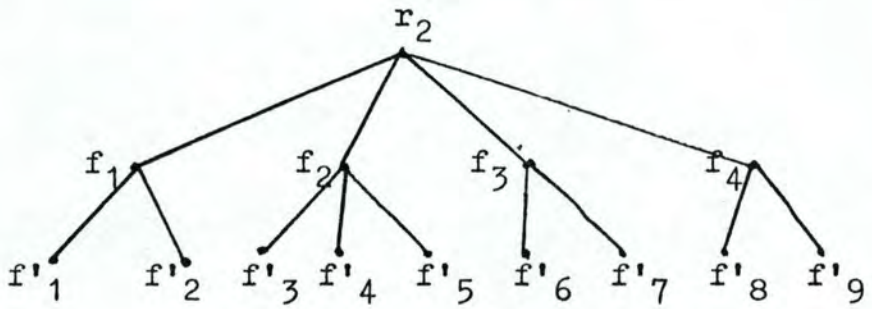


fig. 2

Un arbre est composé d'une racine ou d'une racine et de un ou plusieurs fils.

La figure 1 est un exemple d'arbre constitué d'une racine r_1 et de trois fils f_1, f_2 et f_3.

La figure 2 est un arbre composé de quatre fils; ceux-ci (f_1, ..., f_4) admettent à leur tour des fils (f'_1 et f'_2 pour f_1) : on dira alors que f_1 est un noeud. Le premier fils (numérotation de gauche à droite) de r_2, par exemple, sera considéré comme un sous-arbre de racine f_1.

Sauf mention spéciale, un fils peut ou non à son tour comporter un ou plusieurs fils. La notion de noeud sous-entendra l'existence obligatoire de fils.

Un arbre est typé suivant le type de sa racine.

On parlera d'une instance d'arbre de type T' comme exemplaire d'un arbre de type T'.

Un univers U est un ensemble d'instances d'arbre d'un type déterminé.

De même, on définit un sous-univers comme un sous-ensemble d'un univers U déjà défini.

Un fils (ou racine) f est dit père de f' si f admet f' comme fils.

On dira qu'un fils f admet un frère f' si le père de f admet f' comme fils aussi. On parlera aussi du frère droit et du frère gauche. Ainsi, dans la figure 2 page 18 , f'_4 est le frère droit de f'_3 et f'_6 est le frère gauche de f'_7.

Un constructeur est une fonction permettant de fournir des instances d'arbre d'un type déterminé.

Un fils d'un arbre n'admettant pas de fils est dit fils terminal ou encore feuille.

Nos instances d'arbre seront construites d'une façon incrémentale. On construit, par exemple, une instance d'arbre à zéro fils appartenant à un type T_1 et une autre appartenant à un type T_2. Ensuite, on peut construire une instance d'arbre appartenant à T_3 avec deux fils. Les deux fils sont par exemple les instances fournies précédemment. On dira alors qu'on dispose d'une instance d'arbre de type T_3 ayant deux fils qui ont respectivement comme type T_1 et T_2. Sauf mention explicite, quand on parlera de type d'instance d'arbre, c'est celui de la racine qui sera pris en considération.

Déclaration des univers et des constructeurs

. (defuniverse CATEGORIE)

Cette "instruction" a pour effet de déclarer l'univers appelé CATEGORIE.

. (defuniverse COUPLE ~ CATEGORIE)

L'univers COUPLE est déclaré comme sous-univers (sous-ensemble) de l'univers CATEGORIE.

. (defuniverse HOMME)
  (defuniverse FEMME) } On déclare HOMME et FEMME comme univers.

. (defcons f-instance-homme ~ HOMME)

On définit le constructeur f-instance-homme appartenant à l'univers HOMME. Cette "instruction" a pour effet de générer une fonction Lisp qui produit des instances d'arbre appartenant à l'univers HOMME. Cette fonction est sans arguments.

. (defcons f-instance-femme ~ FEMME)

Cf. le commentaire précédent où HOMME est remplacé par FEMME.

. (defcons f-instance-couple ~ COUPLE (HOMME,FEMME))

On définit le constructeur f-instance-couple. Cette ligne a pour effet de générer une fonction Lisp à deux arguments. Les deux arguments à fournir sont des instances d'arbre appartenant respectivement à l'univers HOMME,FEMME. En fournissant les deux arguments, la fonction générée produit une instance d'arbre appartenant à l'univers COUPLE et ayant comme fils les arguments fournis. Les deux fils appartiennent respectivement à l'univers HOMME,FEMME.

Les différentes possibilités de CEYX n'ont pas été présentées. Notons que les instances d'arbre peuvent avoir zéro ou plusieurs fils. Ces fonctions générées produisent aussi des instances d'arbre de zéro ou plusieurs fils. Ainsi dans (defcons f-instance-homme ~ HOMME), la fonction générée produit des instances d'arbre à zéro fils. Par contre, dans (defcons f-instance-couple ~ COUPLE (HOMME,FEMME)), la fonction générée produit des instances d'arbre à deux fils (car on spécifie entre parenthèses HOMME,FEMME).

Autres fonctions

Supposons que l'on dispose d'une instance d'arbre x d'un type déterminé et ayant par exemple deux fils.

. La fonction (GETSON x n) renvoie le n^{e} fils (numérotation de gauche à droite, n > 0) s'il existe, sinon la valeur NIL.

. La fonction (PUTSON x n y) où x,y : instances d'arbre et n : entier > 0 a pour effet de remplacer le n^{e} fils de l'instance d'arbre x par l'instance d'arbre y.
Le remplacement se fait sous les deux conditions suivantes : l'arbre x possède un n^{e} fils et l'univers de x est le même que celui de y ou l'univers de y est un sous-univers de x.
La fonction renvoie l'instance d'arbre x après substitution.
Si l'une des deux conditions n'est pas vérifiée, une erreur est signalée et la substitution n'a pas lieu.

"Actions sémantiques"

A chaque constructeur on peut associer des actions dites sémantiques. Ces actions ne sont rien d'autre que des fonctions Lisp, elles sont regroupées en fonction Lisp (une fonction par constructeur).

La fonction (SEND x) où x est une instance d'arbre

Cette fonction a pour effet d'appeler la fonction (les "actions sémantiques") associée au constructeur de chaque noeud (ou fils) de x. Le "parcours" de x est à la charge de l'utilisateur.

1.2.3. LEYACC

Partant d'une définition de la grammaire d'un langage, ce logiciel génère un analyseur syntaxique pour ce langage.
Il s'agit d'une version Lisp du logiciel bien connu YACC écrit en C sous Unix. Il n'existait malheureusement pas d'équivalent de LEX. Ainsi l'analyse lexicale est à charge de l'utilisateur.

"Comment localiser le vénérable et secret hexagone qui l'abritait ? Une méthode rétrograde fut proposée : pour localiser le livre A, on consulterait au préalable le livre B qui indiquerait la place de A; pour localiser B on consulterait au préalable le livre C et ainsi de suite jusqu'à l'infini ... C'est en de semblables aventures que j'ai moi-même prodigué mes forces, usé mes ans."

Jorge Luis Borges,
La Bibliothèque de Babel
in Fiction

CHAPITRE 2 PRESENTATION SUCCINCTE DE LA BOITE A OUTILS

Dans ce chapitre, nous exposerons d'abord quelques concepts de base relatifs à l'édition syntaxique, ensuite nous présenterons succinctement la boîte à outils. Nous terminerons ce chapitre en donnant l'architecture de cette dernière conçue et réalisée ainsi que la philosophie du développement de cette architecture.

2.1. INTRODUCTION

Rappelons que notre travail s'insère dans les outils logiciels d'aide à la spécification, conception et mise au point de programmes.

Rappelons aussi que l'objet de notre travail est de réaliser un éditeur syntaxique pour le langage SPES. Ne disposant pas d'un générateur d'éditeur syntaxique de type MENTOR, il nous incombe de concevoir la tâche à partir d'outils de base (CEYX en particulier, cf. chapitre 1, p.18).

En réalité, l'intérêt est apparu de concevoir non seulement un éditeur syntaxique mais aussi un ensemble d'outils généraux nécessaires à sa mise en place et pouvant également être utilisés dans d'autres contextes. Notre souci est en effet de concevoir un système extensible et non figé. Avec de tels outils nous laisserons le soin à l'utilisateur de les assembler, les composer pour couvrir ses propres besoins.

2.2. GENERALITES SUR LA BOITE A OUTILS

Dans la section relative à CEYX (chapitre 1, page 18) nous avons vu certaines possibilités de ce logiciel, notamment la génération d'instances d'arbre. Comme on pourra le remarquer au paragraphe suivant la structure des données utilisée pour les éditeurs syntaxiques est l'arborescence. Dès lors il est apparu nécessaire de se doter des moyens pour la gestion des arbres. C'est ainsi qu'on a crée des fonctions (ou primitives) permettant de parcourir une instance d'arbre, d'insérer ou de supprimer un fils d'une instance d'arbre, de récolter une instance d'arbre, ... Toutes ces fonctions et d'autres permettront la mise en place de l'éditeur syntaxique pour le langage SPES. D'autres fonctions (primitives) ont été créer pour prôduire un diagnostic (résultat d'analyse d'un ou plusieurs énoncés) sur la forêt d'instance d'arbre.

2.3. L'EDITION SYNTAXIQUE

Des études approfondies sur ce sujet peuvent être trouvées dans [DON,79], [TEI,81]. Nous nous contenterons d'exposer quelques concepts de base relatifs aux éditeurs syntaxiques.

2.3.1. Introduction

En toute généralité, on peut dégager quatre fonctions d'édition : la création, l'insertion, la suppression et la visualisation. Dans un éditeur habituel ces fonctions opèrent sur le texte vu comme un ensemble non structuré de lignes. Dans le cas des éditeurs syntaxiques ces fonctions opèrent sur des unités syntaxiques du langage. Dans la plupart des cas, chaque constuction correspond à une production de la grammaire considérée. Cette relation entre constructions et productions de la grammaire fait naître une fonction de contrôle pour resteindre le domaine d'application des fonctions de création, d'insertion et de suppression. Ainsi l'univers du fils à insérer doit être compatible avec celui de l'instance d'arbre où il va être inséré (cf. les préconditions de Put-son dans le chapitre 1, page 21). Ou encore la suppression d'un fils obligatoire (fonction de la connectivité) est interdite. Ces restrictions et d'autres sont à la charge de la fonction de contrôle. La forme la plus naturelle et la plus employée pour la représentation en mémoire des différentes constructions syntaxiques est l'arbre. L'exemple qui va suivre nous illustre ce qui précède et nous permet d'introduire quelques concepts.

Considérons les productions

IF →"if" CDT "then" L-S "else" L-S
CDT →var "<" var
L-S →L-S ";" S
S →var ":=" var

A ces productions, on peut associer les constructions génériques (C-G en abrégé) suivantes :

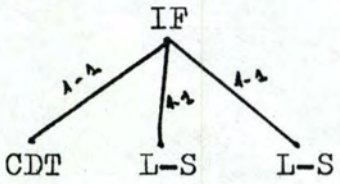


fig. 1

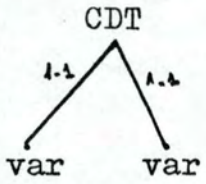


fig. 2

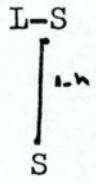


fig. 3

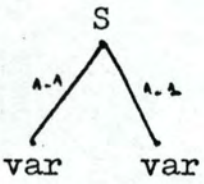


fig. 4

Considérons maintenant la chaîne suivante :

if a b then a:=c else b:=d

Cette dernière aura la construction associée

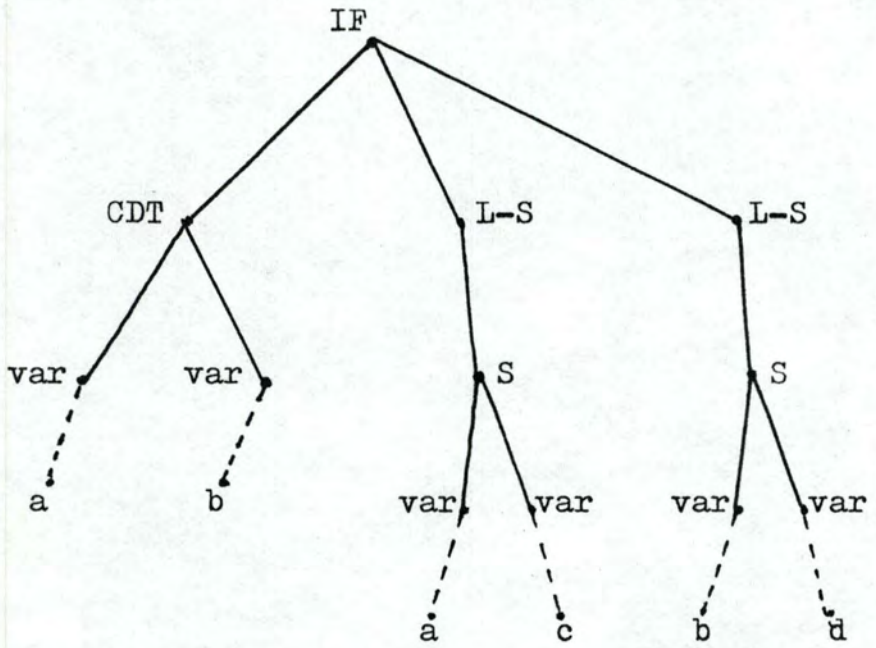


fig. 5

2.3.2. Remarques

- Une C-G n'est qu'une représentation schématique d'une production ou seuls les non-terminaux sont pris en considération.

- Selon la connectivité indiquée sur l'arc, on peut remarquer qu'il existe des C-G admettant un nombre quelconque d'arcs (connectivité 1-n), d'autres admettant un nombre fixe (connectivité 1-1). Les premières sont dites à arité variable, les deuxièmes à arité fixe.

- Les points de chaque C-G sont étiquetés. Ainsi, la C-G de la figure 1 possède l'étiquette IF, les points aux extrémités des trois arcs possèdent respectivement les étiquettes CDT, L-S et L-S.

- La construction associée (figure 5) n'est qu'une composition des C-G (figures 1 à 4). Cette composition n'est pas arbitraire, les étiquettes jouent le rôle d'articulation.

- Les arcs en trait discontinu aux extrémités de la construction associée (figure 5) représentent des propriétés (ou attributs). Considérons une instance d'arbre représentant cette construction associée d'une instance d'arbre. On aura alors des fonctions (ou primitives) pouvant lire et changer cette propriété attachée à un fils quelconque de cette instance d'arbre.

- Partant d'une instance d'arbre représentant la construction associée (figure 5) la fonction de visualitation aura pour effet de la parcourir, de récolter les propriétés et d'ajouter ponctuation, mots-clé, indentation afin de fournir un texte dans une forme agréable.

- A l'état actuel, il existe des systèmes générateurs produisant un éditeur syntaxique pour un langage à partir de la description dans son formalisme par une syntaxe de type BNF.

2.4. ARCHITECTURE GLOBALE DE LA BOITE A OUTILS

Certaines définitions introduites (p. 18) et quelques concepts de la section précédente sont inhérents à l'édition syntaxique. Leur introduction sera utile pour la compréhension du troisième chapitre.

Nous avons également choisi la structure arborescente comme représentation interne de nos spécifications (ou énoncés) en langage SPES. Contrairement à certains systèmes conçus en vue de l'édition syntaxique, on n'impose pas dans notre cas de racine commune à toutes les instances d'arbre. L'inconvénient majeur d'avoir une seule racine commune oblige l'utilisateur à raccrocher toutes ses instances d'arbre. En pratique, il arrive qu'on ne veuille pas ou qu'il ne soit pas possible de raccrocher immédiatement une instance d'arbre. Le système à une seule racine oblige à un raccrochement même momentané. Dans notre cas, on permet plusieurs racines; ainsi l'utilisateur dispose d'une forêt d'arbres. Nous avons alors identifié un certain nombre d'opérations générales de manipulations d'arbres que l'on est amené à appliquer dans ce contexte; nous avons structuré ces opérations en niveaux où la relation définie entre niveaux est : le niveau i+j utilise le niveau j ($\forall j>0$).

Nous disposons actuellement de quatre niveaux, un niveau étant un ensemble de familles, chacune étant composée en fonctions et réalisant une tâche particulière.

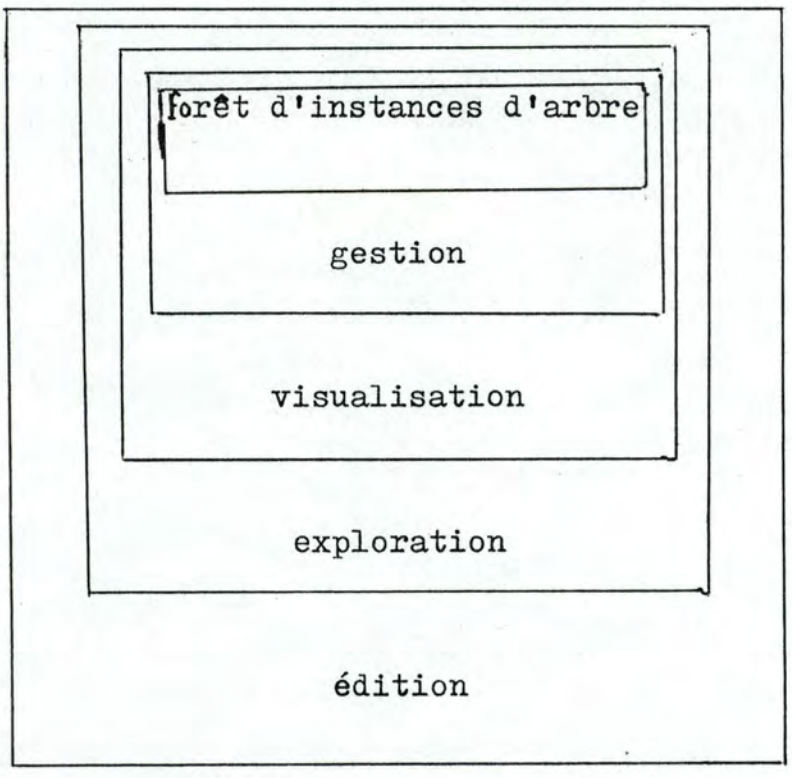
Le premier niveau rassemble les familles de gestion d'instances d'arbre (construction et parcours d'instances d'arbre, insertion, suppression et remplacement d'un fils d'une instance d'arbre), de vérifications simples ("filetage" des propriétés d'instance d'arbre) et de familles produisant des informations utiles concernant la forêt d'arbres.

Le deuxième niveau rassemble la famille de visualisation d'instances d'arbre et enfin celle de paramétrage. Ces dernières permettent d'obtenir des récapitulatifs.

Le troisième niveau rassemble des fonctions permettant l'exploration des instances d'arbre afin de fournir un diagnostic (résultat d'analyse d'un ou plusieurs énoncés) sur la forêt d'instances d'arbre. Ainsi on y trouve des fonctions produisant une "cross-reference" pour un objet ou pour un énoncé, des fonctions pour trier topologiquement le graphe relationnel des objets d'un énoncé, le graphe relationnel des références entre énoncés, etc.

Le quatrième niveau contient les fonctions d'édition proprement dites.

Comme il a été indiqué, la philosophie de cette découpe est : le niveau i+j utilise le niveau i $(i,j>0)$. Ainsi, au niveau 2 on ne s'occupera plus du déplacement (ou parcours d'instance d'arbre) puisque celui-ci est assumé par le niveau 1.
Le schéma suivant peut être suggéré:



Les quatre niveaux ci-dessus nous ont paru indispensable à tout environnement de manipulations de spécifications formelles. D'autres niveaux pourront éventuellement y être ajoutés au fur et à mesure que la nécessité s'en fera sentir. Dans le contexte d'un projet de recherche où les perspectives peuvent fluctuer une telle approche incrémentale ("bottom-up process") nous a paru réaliste.

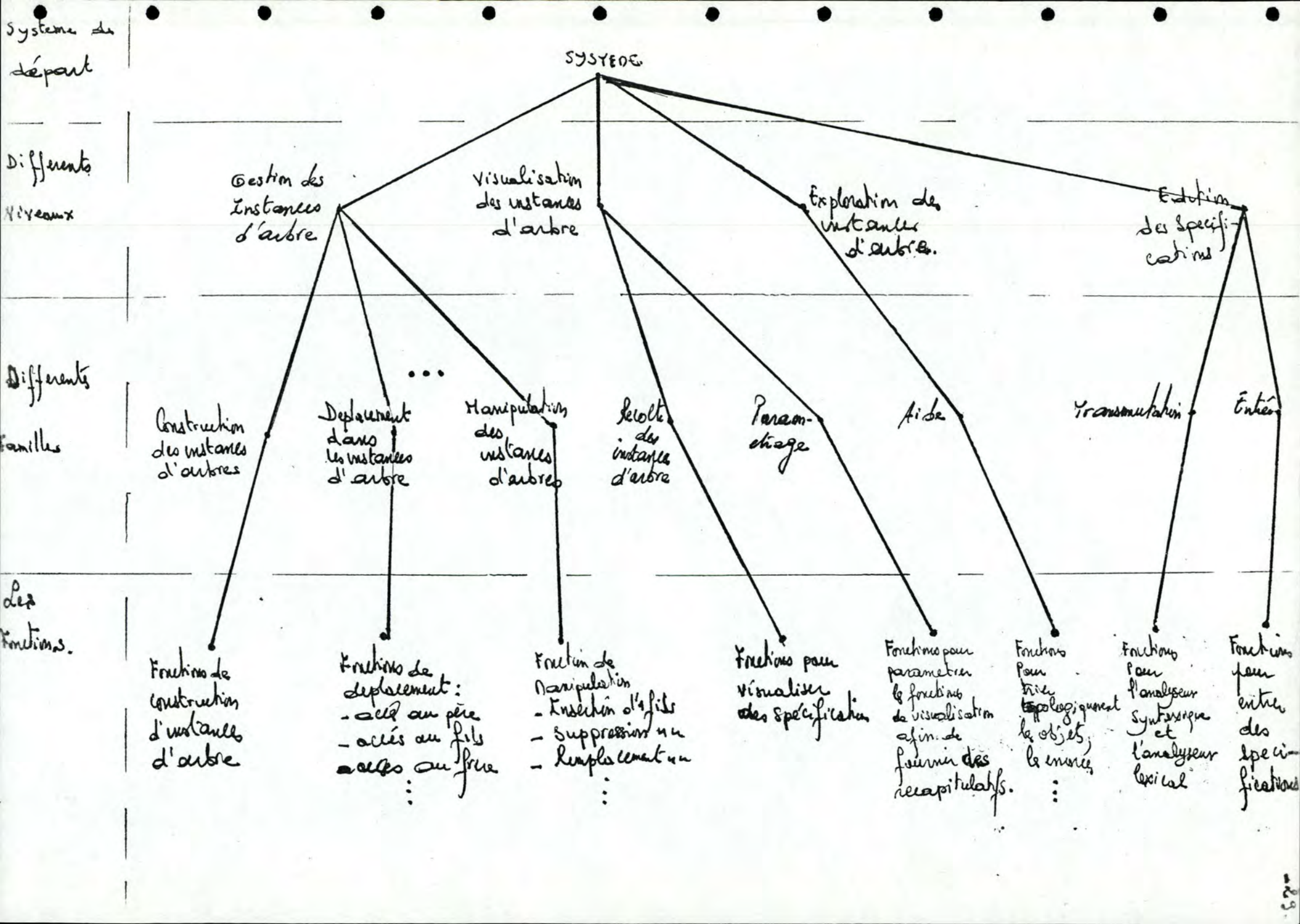
Comme nous l'avons dit, chaque niveau remplit une tâche; chaque niveau est lui-même subdivisé en familles de fonctions. Par exemple, au premier niveau on retrouve une famille permettant la création d'instances d'arbre tandis qu'une autre s'occupera de les parcourir. Enfin chaque famille est décomposée en fonctions. Certains critères nous ont servi pour la décomposition d'une famille en fonctions :

. La non redondance des fonctions. Aucune fonction ne peut être obtenue par composition d'autres fonctions. Ceci nous garantit la minimalité du système obtenu;

. Le couplage minimal entre fonctions. Les seuls interfaces sont les instances d'arbre. On sait qu'un fort couplage pose un problème épineux lors d'une modification [LAMSW]. Une mise à jour d'une telle fonction entraîne la modification d'autres fonctions;

. Pour certaines familles on assure la complétude des fonctions. La plupart des fonctions recherchées peuvent être obtenues en composant les fonctions existantes. Avec une telle décomposition d'une famille, on aura ainsi une fonction de suppression d'un fils d'une instance d'arbre ou celle d'insertion d'un nouveau fils, etc, toutes ces fonctions appartenant à la famille de "manipulation" d'instances d'arbre.

La discussion ci-dessus peut se résumer dans le schéma de la page suivante.


Système de départ
Différents Niveaux
Différents familles
Les fonctions.
SYSYEDE
Gestion des Instances d'arbre
Visualisation des instances d'arbre
Exploration des instances d'arbres.
Edition des Spécifications
...
Construction des instances d'arbres
Déplacement dans les instances d'arbre
Manipulation des instances d'arbres
Récolte des instances d'arbre
Paramétrage
Aide
Transmutation
Entrée
Fonctions de construction d'instances d'arbre
Fonctions de déplacement :
- accès au père
- accès au fils
- accès au frère
Fonction de manipulation
- Insertion d'un fils
- Suppression d'un
- Remplacement d'un
Fonctions pour visualiser des spécifications
Fonctions pour paramétrer la fonction de visualisation afin de fournir des récapitulatifs.
Fonctions pour trier topologiquement les objets, les ensembles
Fonctions pour l'analyseur Syntaxique et l'analyseur lexical
Fonctions pour entrer des spécifications

En conclusion, on remarquera que la notion d'abstraction a été introduite via les niveaux. Le niveau i+j fait abstraction de problèmes qui peuvent arriver au niveau i (i,j>0). Quant à la notion de modularité, elle a été introduite via les familles. Dans ce contexte, un module est une famille réalisant une tâche particulière. Enfin, la modularité a été poussée plus loin au niveau de chaque famille par sa découpe fonctionnelle.

En suivant une telle démarche, nous espérons que le système bâti recouvrira les avantages(souplesse, extensibilité, ...) des systèmes modulaires.

"il se peut qu'il y ait de vraies démonstrations mais cela n'est pas certain. Ainsi cela me montre autre chose, sinon qu'il n'est pas certain que tout soit incertain, à la gloire du pyrrhonisme"

Pascal,
Pensées, 387

CHAPITRE 3 PRESENTATION APPROFONDIE DE LA BOITE A OUTILS

Ce chapitre présente une étude plus approfondie de la boîte à outils.

3.1. NIVEAU 1 : LA GESTION DES INSTANCES D'ARBRE

Ce niveau se compose de six familles.

3.1.1. Famille 1 : GENERATION DES CONSTRUCTEURS D'INSTANCE D'ARBRE (F1)

Rappelons que la structure de données sous-jacente pour représenter un énoncé est l'arborescence. L'objectif de cette famille est de développer un ensemble de fonctions pour pouvoir construire l'instance d'arbre correspondant à une spécification (ou énoncé) écrite en langage SPES.

C O N C E P T I O N

En présentant l'édition syntaxique nous avions introduit le concept de C-G. Nous avions aussi remarqué d'une part que les C-G peuvent être perçues comme une présentation schématique de la grammaire; d'autre part que la construction associée n'est qu'une composition des C-G. Notre démarche va refléter cette propriété. Tout d'abord nous allons "déduire" les C-G du langage SPES. Ensuite nous associerons à chacune d'entre elles une fonction produisant l'instance d'arbre correspondante. Par la suite (voir chapitre 4), nous verrons qu'au moyen de ces fonctions, il sera possible de construire l'instance d'arbre (arborescence de la construction associée) correspondant à un énoncé en langage SPES.

"Déduction" des C-G

Dans l'exemple donné page 24 (chapitre 2) nous avions "déduit" une C-G par production. En réalité il n'en sera pas toujours ainsi : cette "déduction" n'est pas biunivoque. Ces raisons (union de types) sont en partie inhérents à l'éditeur syntaxique et ne seront pas donnés dans le texte. Le lecteur intéressé peut se référer aux ouvrages traitant de l'édition syntaxique d'une façon approfondie [DON;83] [FRA;81]

Après examen de la grammaire du langage SPES on a "déduit" les C-G de notre langage. Le résultat de cette "déduction" est un ensemble de schémas représentant les C-G.

Association de fonctions

Ayant "déduit" les C-G, il nous fallait produire une fonction par C-G. Chacune de ces fonctions a pour effet de construire une instance d'arbre représentant la C-G à laquelle elle est attachée.

Quelques remarques ont été introduites au sujet des C-G.

- Il existe des C-G possédant un nombre fixe d'arcs; elles sont dites à arité fixe.
- Les C-G qui ont un nombre variable d'arcs sont dites à arité variable.
- Les C-G sont étiquetées. Certaines d'entre elles admettent des propriétés.

L'objectif est de développer une fonction par C-G. Cette fonction produira une instance d'arbre représentant cette C-G. La notion d'arité fixe, variable, d'étiquette ainsi que des propriétés d'une C-G doivent être prises en compte lors du développement de chacune de ces fonctions.

R E A L I S A T I O N

Rappelons qu'au moyen du logiciel CEYX (chapitre 1, page 18), on peut génèer des fonctions (par defcons). Ces fonctions nous produisent des instances d'arbre typées (ou appartenant à un univers). C'est avec ce logiciel que nous allons procéder pour développer ces fonctions.

Ainsi notre tâche pratique consiste, pour chaque C-G, de

-déclarer les univers (le concept d'univers correspond au concept d'étiquette en C-G) au moyen de defuniverse (cf chapitre 1, page 16);

- générer le constructeur (ou fonction) par "defcons" (cf chapitre 1 , page 20). Cette fonction générée par CEYX produira l'instance d'arbre. Celle-ci représente la C-G prise en charge par cette fonction. On dira aussi que la C-G est attachée à cette fonction ou vice-versa.

Après avoir exécuté les "instructions" "defuniverse" et "defcons", le logiciel CEYX génère les fonctions. On dispose

ainsi d'une fonction par C-G.

Le profil de ces fonctions générées est le suivant :

ENTREE	le(s) fils requis par l'instance d'arbre représentant la C-G attachée à cette fonction.
SORTIE	l'instance d'arbre (composée de(s) fils en entrée) représentant la C-G attachée à cette fonction.
PRECONDITION	le type des fils à l'entrée doit concorder avec celui attendu par la fonction. Selon l'arité de la C-G attendue dans la fonction le nombre de fils requis en entrée est soit fixe, soit variable.

Notons enfin qu'on n'a pas encore traité dans cette famille des propriétés de certains fils terminaux. Ce sera un des objectifs de la famille suivante.

3.2.2. Famille 2 : CONSTRUCTION DES INSTANCES D'ARBRE (F2)

Dans la famille 1 nous nous sommes dotés de fonctions permettant de produire des instances d'arbre à chaque C-G. La précondition à ces fonctions générées par CEYX est la concordance entre le type du (des) fils fourni(s) à l'entrée et celui attendu par fonction. Il faut donc vérifier cette précondition; en cas de discordance le logiciel CEYX (version actuelle) signale une erreur à l'interpréteur Lisp qui empile une erreur et produit un message.

Signalons aussi que les propriétés n'ont pas encore été traitées. L'objectif de cette famille est de traiter ces deux problèmes.

C O N C E P T I O N

Traitons tout d'abord le problème de discordance en type. Le moyen d'éviter l'empilement d'erreur par l'intérpréteur Lisp peut être la récupération d'erreur (on condition en Pl/1). Si une discordance est apparue, on prendra soin de la traiter au lieu de la laisser à la charge de l'interpréteur Lisp. Cette pile a une taille finie. Après un nombre fini d'erreurs égal à la taille de la pile on est obligé de la dépiler par la fonction Lisp (Reset). La récupération d'erreur nous garantit d'une part

que le sommet de la pile ne sera pas atteint (par ce genre d'erreurs) ce qui évite à l'utilisateur d'exécuter la fonction Reset. D'autre part, cela permet d'éliminer le message standard de Lisp qui serait émis à un endroit quelconque de l'écran. Ce message peut être remplacé par un autre et visualisé à une ligne précise de l'écran.

Pour résoudre le deuxième problème, une composition de fonctions Lisp et CEYX permettront d'attacher une propriété à un fils terminal donné.

Le 1er problème aurait pu être traité au sein de la famille F1. Il n'en a pas été ainsi pour la raison suivante : les fonctions de F1 sont générées par le logiciel CEYX. On a alors évité de faire un "bricolage" pour les modifier afin de récupérer l'erreur. Ce "bricolage" nous a paru inutile et dangereux.

R E A L I S A T I O N

Récupération d'erreur (*)

Pour chaque fonction appartenant à F1 nous avons crée une autre fonction appelant la fonction générée en F1 et utilisant les fonctions de base Lisp pour récupérer l'erreur. Le profil de ces fonctions est le même que celui des fonctions de F1. Seule la précondition disparaît.

Les fils terminaux à propriété

Ces fonctions produisent une instance d'arbre à un fils en lui attachant une propriété.

Le profil de ces fonctions est le suivant :

ENTREE	la propriété.
SORTIE	l'instance d'arbre possédant un fils avec la propriété. Nil sinon.
PRECONDITION	la propriété doit répondre à certaines règles (cf. la famille suivante F3).

(*) Dans la version actuelle de Lelisp, le problème n'est pas résolu..

3.1.3. Famille 3 : CONTROLE DES PROPRIETES DES FILS TERMINAUX (F3)

C O N C E P T I O N

Certains fils admettent des propriétés. Parmi ces propriétés nous avons cité les identificateurs mais il en existe bien d'autres. Mentionnons par exemple quelques "opérateurs" tels que ($\Rightarrow$, $\Leftrightarrow$) ou encore les constantes numériques et alphanumériques.

La plupart de ces propriétés doivent répondre à certaines règles. Si une règle est violée la propriété ne peut être admise. Un idenficateur doit commencer par un caractère alphabétique et ne peut avoir avoir de caractères spéciaux est un ex. de règle. Il s'agit dès lors de disposer de fonctions permettant de "filtrer" les propriétés.

Différentes méthodes sont envisageables pour la conception de ce "filtre". Notre conception comme pour les autres familles sera fonctionnelle (au sens Lisp). Nous avons inventorié les règles que les propriétés doivent assurer. Ensuite nous avons décomposé les règles en règles élémentaires. Pour chaque fils terminal à propriété nous avons déterminé les règles qu'il doit remplir, en associant une fonction de contrôle à chaque règle élémentaire. Il ne restera plus qu'à composer ces fonctions pour contrôler la propriété d'un fils donné.

R E A L I S A T I O N

Les règles élémentaires

Nous nous bornerons ici à ne donner que quelques exemples de telles règles :

- contrôler si UN caractère est alphabétique;
- contrôler si la longueur d'une chaîne de caractères dépasse m caractères.

A chacune de ces règles élémentaires nous avons associé une fonction. Son profil est le suivant :

ENTREE l'argument que l'on doit contrôler. Ce sera souvent un caractère.

SORTIE t : (vrai) si l'argument répond à cette règle élémentaire;
nil : (faux) si l'argument n'y répond pas.

Les règles

Nous savons qu'une règle n'est qu'une conjonction de règles élémentaires. Aussi la fonction associée à une règle sera une composition des fonctions associées aux règles élémentaires relatées par les opérateurs booléens (AND, OR, etc). Le profil de ces fonctions sera analogue aux fonctions associées aux règles élémentaires. Donnons un exemple. La fonction contrôlant si une propriété tel qu'un identificateur est alphabétique. Cette fonction appelera la fonction associée à la règle élémentaire (contrôler si un caractère est alphabétique) autant de fois que la valeur de la longueur de l'identificateur.

Enfin il ne nous reste plus qu'à créer des fonctions composant les fonctions précitées pour contrôler une propriété d'un fils-terminal. On a par exemple la fonction contrôlant un identificateur, qui vérifie si la longueur dépasse n caractères, si le premier caractère est alphabétique, si la chaîne ne contient aucun caractère spécial, etc. Le profil de ces fonctions reste le même qu'auparavant.

Il est clair que ces fonctions sont appelées par les fonctions (associées aux fils terminaux) de F2, afin de contrôler si la propriété répond aux règles.

3.1.4. Famille 4 : DEPLACEMENT DANS LES INSTANCES D'ARBRE (F4)

A ce stade de l'étude, nous disposons de F1, de F2 et de F3. Elles nous permettent de constuire des instances d'arbre avec quelques vérifications des propriétés.

Un problème se pose alors immédiatement; c'est celui de pouvoir parcourir (ou se déplacer dans) une instance d'arbre. Ce parcours peut être utile pour changer une propriété d'un fils, insérer un nouveau fils, etc.

Parcourir une instance d'arbre est un problème usuel. En disposant de trois fonctions (F, C et P) et de deux prédicats (f et c) notre problème est résolu .

Soit

une instance d'arbre :

x la racine ou un fils quelconque de cette instance d'arbre;

C(x) produit le premier frère droit de x s'il existe. Si x est la racine, son frère n'existe pas (cf. déf. p,19)

F(x) produit le premier fils (le plus à gauche) de x s'il existe.

P(x) produit le père de x si x n'est pas une racine.

les deux prédicats :

f(x) vrai si et seulement si x possède un fils

c(x) vrai si et seulement si x a un frère.

DEFINITIONS

CHEMIN

Si on part de la racine d'une instance d'arbre et qu'on applique avec succès les fonctions présentées F et C un nombre fini de fois, on se retrouve à un fils quelconque de cette instance d'arbre. On appellera CHEMIN entre la racine et ce fils cette suite finie composée de F et C menant de la racine à ce fils.

Par définition, le chemin menant à la racine est vide.

Exemple 1

F C C C F F C C C F

Exemple 2

F F C

Le chemin ℓ de l'exemple 1 montre que partant de la racine on a accédé au premier fils (appelé f1), au frère doit de f1 (appelé f2) puis au frère droit de f2 (appelé f3), ensuite au premier fils de f3, e

On interprétera l'exemple 2 d'une façon similaire.

NIVEAU

On dira que l'on se retrouve au niveau n après avoir parcouru un chemin quelconque ℓ si on compte n occurences de la fonction F dans le chemin ℓ.

Par définition, si le chemin est vide, le niveau est égal à zéro.

C O N C E P T I O N

Dans ce qui précède nous avons exposé le problème en toute généralité; on a posé les fonctions et les prédicats nécessaires pour pouvoir parcourir une instance d'arbre.

A présent, nous allons essayer de développer des fonctions analogues à F, C et P et aux prédicats f et c. Ce développement se fera à partir de la fonction disponible GET-SON de CEYX. Une autre solution de développement consiste à créer des fonctions opérant directement sur la représentation interne des instances d'arbre.

Cette deuxième solution est rejetée pour deux raisons :

. Opérer sur la représentation interne serait en contradiction avec la philosophie d'abstraction décidée pour le développement du projet;

. Bien qu'une telle solution présente l'avantage de la rapidité d'exécution, elle risque d'être coûteuse si une modification de la représentation interne survenait.

Ainsi, notre fonction de base est GET-SON. Rappelons qu'elle permet l'accès au(x) fils immédiat(s) d'une instance d'arbre. Malheureusement, dans CEYX, il n'existe pas de fonction permettant l'accès du fils au père.

La fonction F(x) et le prédicat f(x)

La fonction GET-SON de CEYX combine à la fois la fonction F(x) et le prédicat f(x). La fonction GET-SON est plus générale : en effet, si la fonction F(x) accède toujours au premier fils (s'il existe), GET-SON, lui, peut accéder au n^{e} fils (s'il existe) selon l'argument fourni (cf. chapitre 1, page 21).

Etant donné que nous sommes toujours à la phase "conception" et que nous souhaitons développer des fonctions analogues à F, C, P et aux prédicats f et c, nous dirons que la fonction GET-SON avec les arguments x et 1 (GET-SON x,1) joue le rôle de F(x) et f(x). Il nous reste alors les fonctions C et P et le prédicat c.

La fonction C(x) et le prédicat c(x)

Rappelons que les arguments requis par la fonction GET-SON sont le premier, l'instance d'arbre, le second, le numéro du fils auquel on veut accéder. Si on dispose de la fonction P(x), on peut aisément écrire une fonction FRERE analogue à C(x).

On définit FRERE de la façon suivante :

```
FRERE (x,i)
  P(x)
  GET-SON (x,i+1)
FIN
```

avec

ENTREE x : instance d'arbre
i : le numéro du fils actuel
SORTIE : le frère situé à la position i+1 du fils actuel s'il existe, nil sinon.

De cette manière, le problème est résolu.

La fonction P(x)

Pour résoudre ce problème, deux solutions peuvent être envisagées :
. la première consiste à supposer que la fonction P(x) est suffisamment élémentaire et passer au stade de réalisation;
. la deuxième consiste à développer davantage cette fonction.
Les raisons du développement seront données à la section "Réalisation" (page 46).

Soit - x une instance d'arbre,
- y un fils quelconque de x,
- $\mathcal{C}$ le chemin suivi pour atteindre y,
- n le niveau actuel; il peut se déduire de $\mathcal{C}$ (cf. la définition page 39).
- E une suite vide

Définissons la fonction PERE analogue à P dont la spécification est la suivante :

ENTREE x , y
SORTIE 1. f indique le père de y
2. ℓ indiquera le chemin parcouru de la racine au père de y

L'algorithme suivant réalise cette spécification. Notons f la position courante dans l'instance d'arbre. Cette position est automatiquement mise à jour à chaque déplacement. Les déplacements sont effectués par les fonctions F ou C.

```
PERE (x,ℓ)
  se positionner à la racine de x (notée dans f)
  z ← 1er élément de ℓ
  détruire cet élément de ℓ
  E ← E ∪ {z}
  exécuter z avec l'argument x
  si n (*) = 1
    alors M (ℓ,x)
    sinon M (ℓ,x)
          PERE (x,ℓ)
  fin-si
  ℓ ← E
FIN
```

où M (ℓ,x) sera spécifié plus loin.

A présent, nous allons nous attarder sur le concept du chemin qui est une suite composée de F et C. Notons tout d'abord qu'il est impossible que le premier élément soit C. En effet, le chemin part de la racine et si on applique C(x) (avec x comme racine) la fonction C est sans succès et le premier élément ne peut donc être C. Dès lors, le premier élément est nécessairement F.

Considérons à présent cette suite privée de son premier élément, notée $\bar{\ell}$. Extrayons de $\bar{\ell}$ la sous-suite commençant au premier élément de $\bar{\ell}$ et s'arrêtant au prochain élément F, celui-ci n'appartiendra pas à cette sous-suite. Si le premier élément de $\bar{\ell}$ est F, cette

(*) n étant le niveau, il sera déduit à partir de ℓ.

sous-suite est considérée comme vide. Il est clair que cette sous-suite ne contient aucun élément F. Nous noterons la longueur d'une sous-suite (resp. d'un chemin) L (resp. L_ℓ). La longueur est définie comme le nombre d'éléments de cette sous-suite (resp. chemin). Si cette sous-suite (resp. chemin) est vide, la longueur est égale à zéro. Sau mention contraire, quand on parlera de sous-suite, il s'agira de la sous-suite extraite (ou formée) du chemin ℓ.

Considérons le chemin ℓ: F C G F on a :

la longueur de ℓ notée L_ℓ est 4

$\bar{\ell}$ = G C F

et la sous-suite sera C C et sa longueur (notée **L**) est 2.

Spécifions à présent la fonction M (ℓ,x).

ENTREE ℓ le chemin suivi de l'instance d'arbre considérée
x l'instance d'arbre

SORTIE t
nil

Soit
- m le niveau à l'entrée de la fonction
- m' le niveau à la sortie de la fonction
- ℓ le chemin à l'entrée de la fonction
- L_ℓ la longueur du chemin en entrée
- ℓ' le chemin à la sortie de la fonction
- L'_ℓ la longueur du chemin en sortie
- f la position courante en entrée de la fonction
- f' la position courante à la sortie de la fonction
- i le numéro du fils à la position courante à l'entrée de la fonction
- i le numéro du fils à la position courante à la sortie de la fonction
- S la sous-suite extraite de ℓ
- L la longueur de cette sous-suite S

Après exécution de M, on a les post-conditions suivantes à respecter :

$m = m'$ (1)

$L'_{\mathcal{C}} = L_{\mathcal{C}} - L$ (2)

$i = i + L$ (3)

E et f (la position courante) sont consdiérés connus par la fonction M. L'este des variables définies ci-dessus peuvent se déduire de $\mathcal{C}$ et de f.

L'algorithme suivant réalise la spécification ci-dessus.

```
AM (C,x)
  z ← 1er élément de C
  répéter tant que (z ≠ F et (C est non vide))
    exécuter z avec l'argument x
    E ← E ∪ {z}
    détruire le 1er élément de C
  fin-répéter
FIN
```

Vérifions que cet algorithme respecte les post-conditions spécifiées.

(1) $m = m'$

Ici, on exprime que la fonction ne doit pas changer le niveau. Notons que seule l'exécution de la fonction F change le niveau. Or la bouche de cet algorithme empêche toute exécution de F.

(2) $L'_{\mathcal{C}} = L_{\mathcal{C}} - L$

Ici, on exprime que la fonction doit effectuer la destruction des éléments du chemin qui ont formé la sous-suite extraite de ce chemin.

Notons que la boucle de cet algorithme formé bien cette suite. En effet, l'algorithme PERE détruit le 1er élément de $\mathcal{C}$; on obtient ainsi l'équivalent de $\vec{\mathcal{C}}$. Ensuite on boucle dans l'algorithme AM pour former de façon incrémentale cette sous-suite. A chaque passage dans la boucle on détruit un élément de cette sous-suite ce qui garantit que la longueur du chemin diminue de L éléments.

(3) $i' = i + L$

Ici, on exprime que, si à l'entrée de l'algorithme, la position courante indique le fils numéro i, à la sortie, elle indiquera le fils numéro i + L.

La sous-suite étant formée dans la boucle, tous les éléments de cette sous-suite sont nécessairement C et la longueur de cette sous-suite est L. Au cours de chaque passage dans la boucle, l'ordre "exécuter z avec l'argument x" fera passer la position du fils n° i au fils n° i+1. Ceci nous garantit que la position sera i + L.

Dans ce qui suit, nous allons étudier l'algorithme réalisant la fonction PERE.

Notons que si y appartient au niveau i, le père de y appartient au niveau i - 1 ($i>0$). A l'algorithme PERE l'opération "détruire z" (où z est nécessairement F) décrémente le niveau de 1. A chaque passage d'un niveau à un autre, on exécute l'algorithme AM et la post-condition (3) est respectée, ce qui nous garantit que le chemin précédemment parcouru d'un niveau à un autre est reparcouru.

L'algorithme PERE s'arrête quand la condition n=1 est satisfaite c'est-à-dire quand le niveau i-1 est atteint. D'où le niveau du père de y est atteint.

Avant de sortir définitivement de l'algorithme PERE, on exécute l'algorithme AM et la post-condition (3) est respectée ce qui nous garantit que f (la position courante) indique le père de y.

La suite E initialement vide et l'opération "ℓ-E" assure la sortie (2) de la spécification de la fonction PERE.

A présent, déterminons si les deux algorithmes s'arrêtent après un temps fini.

L'algorithme de la fonction PERE s'arrête quand le niveau est égal à 1.

La première fois que l'on entre dans l'algorithme, le niveau (*) est nécessairement ≥ 1.

1er cas : niveau (*) = 1

on exécute l'algorithme de la fonction M puis on s'arrête.

2e cas : niveau (*) $>$ 1

dans ce cas, notons que :

- l'opération "détruire z" décrémente le niveau;
- l'algorithme AM ne modifie pas le niveau (cf. (3) page 43)

(*) Ce niveau est déduit de l'argument ℓ en entrée.

Ainsi l'algorithme associé à la fonction PERE ne peut s'appeler indéfiniment et on passera nécessairement par le premier cas.

La terminaison de l'algorithme AM est evidente.

R E A L I S A T I O N

Lors de l'introduction de cette famille, nous avons introduit trois fonctions F, C et P. A la conception on a proposé des fonctions (et algorithmes) pour effectuer F, C et P.

Remarquons que l'algorithme pour réaliser la fonction C utilise la fonction P. Or l'algorithme proposé pour effectuer la fonction P utilise aussi la fonction C d'où une situation d'interblocage.

Rappelons qu'à la conception on a proposé deux solutions possibles pour réaliser la fonction P :
- on la considère comme élémentaire;
- on la développe davantage.

Pour supprimer cet interblocage, on développera deux versions de la fonction P(x). La première version applique la première solution, la seconde la seconde version. Il est clair que la première version ne doit pas utiliser la fonction C.

Pour réaliser cette première solution on utilisera un deuxième pointeur(*) qui suivra le pointeur courant. Ce deuxième pointeur est positionné sur le père du fils désigné par le pointeur(*) courant.

Exposons maintenant, comme prévu, la raison du développement de la deuxième solution.

Les algorithmes proposés pour réaliser F, C et P consistent à accéder respectivement au premier fils de gauche immédiat, au premier frère de droite immédiat et au père immédiat. En pratique, il arrive que l'on veuille accéder non seulement au père immédiat mais aussi au père du père ... de ce père (la même nécessité existe au niveau des fils et des frères).

Pour satisfaire la nécessité qui vient d'être mentionnée, on a paramétré les fonctions F, C et P. On comprend dès lors la raison du refus d'appliquer la première solution à la fonction P.

Notons enfin que la fonction C permet d'accéder au frère droit. Si on veut atteindre le frère gauche, il suffit d'accéder au père puis d'accéder, par les fonctions F, au premier fils , ensuite, accéder successivement aux différents frères de ce fils, jusqu'à l'accès souhaité.

(*) Notion équivalente à la notion de position introduite à la conception.

Afin d'éviter à l'utilisateur d'accéder au père on a introduit une quatrième fonction G qui accède au fils de gauche. L'algorithme réalisant cette fonction se conçoit de façon similaire à celui de la fonction C. L'introduction de cette fonction G modifie la définition du CHEMIN : aux éléments initiaux F et C composant la suite, on ajoute un troisième élément, l'élément G. Ces quatre fonctions F, C, P et G sont regroupées en une fonction à trois paramètres

DEPLACER (m, x, n)

avec le profil suivant :

PRE-CONDITION x doit être une instance d'arbre

ENTREE
- m un mnémonique
- x une instance d'arbre
- n entier supérieur ou égal à zéro

SORTIE
si opération réussie, t : la position courante indiquera le fils ou le frère (droit ou gauche) ou le père désiré dans l'instance d'arbre selon les arguments m et n fournis;
sinon, nil.

EFFET
suivant m précisant l'opération à faire
la fonction répète l'opération n fois
(n > o dans ce cas).
Le mnémonique peut être :
- G : pour la fonction C
- D : pour la fonction D
- H : pour la fonction P
- B : pour la fonction F

Relevons les cas particuliers :
- n=0 et m = G : accès au frère le plus à gauche
- n=0 et m = D : accès au frère le plus à droite
- n=0 et m = H : accès à la racine de x
- n=0 et m = B : accès aux feuilles de x

Remarque

A chaque instance d'arbre on a attaché une variable indiquant la position courante. Cette variable est transparente à l'utilisateur. La position courante est mise à jour à chaque exécution de la fonction DEPLACER.

Les fonctions d'information

Lors de la conception de la fonction P, nous avons supposé que nous disposions du chemin parcouru et du niveau de l'instance d'arbre considéré. Outre ces deux informations, on aura besoin en pratique d'autres renseignements. Les fonctions suivantes répondent à ce besoin.

A. La fonction ITINERAIRE

Cette fonction fournit le chemin le plus court (cf. définition plus bas) parcouru. L'utilité de cette fonction sera donnée plus loin.

Rappelons que les éléments de la suite composant la suite d'un chemin ne sont plus seulement F et C mais F, C et G.

Définition du chemin le plus court

Un chemin ℓ conduisant de la racine d'une instance d'arbre x à la position courante de celle-ci est dit le plus court (ou minimal) si parmi tous les chemins possibles $\ell_1, \ldots, \ell_n$ menant de la racine de x à sa position courante on a $L_\ell < L_{\ell_i}$ $i \in 1, \ldots, n$ où L_ℓ est la longueur du chemin ℓ.
et $L_{\ell_1} \ldots L_{\ell_n}$ sont respectivement les longueurs des chemins $\ell_1 \ldots \ell_n$.

L'introduction de la fonction G peut entraîner des déplacements susceptibles d'allonger le chemin. En effet, considérons que la position courante indique le fils noté s. Si on applique la fonction C pour se retrouver au premier frère droit de s, ensuite la fonction D, on se retrouve de nouveau sur le fils de départ s. Ces déplacements allongent inutilement le chemin. La fonction ITINERAIRE annule ces opérations allongeant le chemin initialement construit pour nous fournir le chemin minimal.

Remarquons que la détermination du chemin le plus court augmente la vitesse d'exécution de l'algorithme proposé par la fonction PERE.

Notons enfin qu'il existe d'autres utilisations de la fonction ITINERAIRE. Citons par exemple le cas suivant : ayant déjà parcouru un chemin, on voudrait parcourir un autre chemin puis reparcourir l'ancien.

ITINERAIRE x

avec le profil suivant :

ENTREE x une instance d'arbre

SORTIE le chemin le plus court parcouru pour atteindre la position courante de cette instance d'arbre x
nil : si x n'est pas une instance d'arbre ou si le chemin est vide.

B. La fonction TYPE

Cette fonction produit le type ou univers du fils (ou racine) à la position courante. L'utilité de cette fonction sera donnée plus loin.

Nos instances d'arbre représentent des constructions associées à des spécifications. Nous savons qu'une construction associée est une composition de C-G et qu'une instance d'arbre est construite de façon incrémentale, chacun des fils de cette instance d'arbre étant lui-même une instance d'arbre (cf. chapitre 1, page 19).

Il arrive alors qu'après un chemin quelconque dans une instance d'arbre, on veuille connaître le type du fils à la position courante (ou de la racine si le chemin est vide). La fonction TYPE a été créée dans ce but :

TYPE x

avec le profil suivant :

ENTREE x une instance d'arbre

SORTIE le type de ce fils (ou de la racine si le chemin est vide)

Cette fonction peut être utilisée dans différents contextes, par exemple :

. D'après le type du fils (ou de la racine), on pourrait "déduire" sa structure (son arité, terminal ...)

. On pourrait aussi savoir s'il est permis de faire telle opération. La famille qui suit précisera ces opérations.

C. La fonction ETAT

Après un chemin quelconque dans l'instance d'arbre considérée, l'invocation de la fonction ETAT fournit les informations suivantes :

. le niveau de l'instance d'arbre après ce chemin;
. le nombre de fils (ou la racine si le chemin est vide) indiqué par la position courante;
. la classe (fixe, variable, terminal, à propriété);
. le numéro du fils (si le chemin est non vide) par rapport à son père;
. la classe du(des) fils si ce fils est à arité fixe .

ETAT x

ENTREE x est une instance d'arbre

SORTIE la liste informative
nil si x n'est pas une instance d'arbre.

Ces informations sont très utiles pour avoir une idée de l'état de chacune des instances d'arbre de la forêt.

D. La fonction LISTER

Rappelons que, contrairement à certains systèmes conçus en vue de l'édition syntaxique, on n'impose pas une racine commune à tous les arbres. Etant donné l'existence de plusieurs racines, on a alors crée la fonction LISTER qui liste toutes les racines de la forêt d'arbre.

E. La fonction DETRUIRE

Ayant plusieurs racines, il est commode de procéder à un "nettoyage". On pourra ainsi détruire quelques instances d'arbre. C'est l'objectif de la fonction DETRUIRE.

DETRUIRE x

ENTREE x est une instance d'arbre

SORTIE t si la destruction (physique) a été effectuée;
nil si cette instance d'arbre n'existe pas.

3.1.5. Famille 5 : MANIPULATION D'INSTANCES D'ARBRE (F5)

Outre la construction d'instances d'arbre et leur parcours, il est fort utile de disposer de fonctions réalisant, par exemple, l'insertion ou la destruction d'un fils à une instance d'arbre. C'est dans cette famille que l'on va traiter ces problèmes.

En toute généralité, on pourra poser le problème de la façon suivante : étant donné une instance d'arbre, quelles sont les opérations que l'on peut lui appliquer ?

Habituellement, on peut relever deux opérations de base : l'insertion d'un nouveau fils ou la suppression d'un fils dans une instance d'arbre.

Avant de procéder à la conception et à la réalisation de ces opérations, nous allons nous occuper davantage de la nature de notre structure d'instance d'arbre. A cet effet, rappelons que l'on est en présence de deux classes de C-G : celle à arité fixe et celle à arité variable. Les instances d'arbre qui représentent ces C-G sont également de deux classes. Examinons séparément ces deux cas.

Première classe : Arité fixe

Dans ce cas, les instances d'arbre de cette classe ont un nombre fixe de fils. Ces deux opérations (insertion et destruction) sont interdites puisque, après exécution de l'une d'elles, le nombre de fils varie. Par contre, si l'on considère la combinaison de ces deux opérations pour en faire une opération atomique, le nombre de fils restera inchangé et l'opération est permise.

Deuxième classe : Arité variable

Dans ce cas, les deux opérations (insertion et destruction) sont permises.

C O N C E P T I O N

Avant d'aborder les algorithmes réalisant les fonctions d'insertion, de suppression et de remplacement (opération atomique combinant la suppression et l'insertion), remarquons que les opérations

secondaires suivantes peuvent être très utiles à l'exploitation.

La copie d'un fils d'une instance d'arbre produit une instance d'arbre dont la racine est le fils. Cette copie peut servir à la duplication. Elle peut aussi constituer un avantage : la copie d'un fils forme à partir de celui-ci une instance d'arbre qui sera plus rapide à parcourir, plus facile pour réaliser des insertions, des suppressions, etc. En fin de traitement, on peut remettre cette copie modifiée à la place du fils copié, duplicier cette copie modifiée. Les fonctions Lisp peuvent résoudre ce problème de copie.

Rappelons que, parmi les fils, il existe des fils terminaux à propriétés. A cet effet, la famille F3 contient des fonctions produisant des instances d'arbre à un fils avec la propriété attachée. Pour modifier la propriété d'un fils terminal à propriété d'une instance d'arbre, deux solutions sont possibles :

- la première consiste à créer une instance d'arbre à un fils avec la nouvelle propriété et remplacer l'ancien fils terminal par cette nouvelle instance d'arbre;
- la deuxième consiste à disposer de fonctions particulières opérant directement sur les propriétés.

C'est cette deuxième solution qui sera retenue vu sa rapidité d'exécution et sa facilité pour l'utilisateur. Ainsi, en plus de l'opération de lecture (cf. chapitre 2, page 25), on aura celle d'écriture pour modifier une propriété. Les fonctions Lisp et CEYX peuvent résoudre ce problème.

A présent, on va développer la fonction de remplacement et les deux fonctions d'insertion et de suppression.

La première existe en CEYX: c'est la fonction PUTSON (cf. chapitre 1, page 21). Reste alors les deux autres fonctions. Dans les ouvrages traitant des arbres il existe plusieurs algorithmes (récursifs ou non) d'insertion et de suppression. Notre solution pour résoudre ces problèmes sera différente.

On pourrait poser le problème d'insertion (resp. de suppression) de la manière suivante.

Etant donné une instance d'arbre à n fils, on voudrait passer à une instance d'arbre à n+1 (resp. n-1) fils.

On peut remarquer qu'on a fait abstraction du numéro du fils à insérer (supprimer) ou de la position à laquelle le nouveau fils doit être inséré (supprimé). On verra que la prise en compte de ce fait n'influence pas profondément la solution proposée.

Pour résoudre le problème, nous allons exploiter les fonctions de la famille F2 (ou F1). On recopie les fils eixstants, on ajoute le nouveau et on reconstruit une nouvelle instance d'arbre à n+1 fils.

Puisqu'on peut appliquer l'opération d'insertion ou de suppression sur l'instance d'arbre initiale, celle-ci est certainement à arité variable. On sait en outre que la fonction qui a construit cette instance d'arbre initiale appartient à F1 (ou à F2). Cette fonction admet aussi un nombre quelconque d'arguments (ou de fils) en entrée. Ainsi la reconstruction se résume en les points suivants :

- recopier les n fils existants;
- préparer la liste d'arguments de ces n fils et du nouveau fils à insérer.

Rappelons (cf. chapitre 1, page 20) que l'ordre des arguments est important. En effet, considérons la fonction f (acceptant un nombre d'arguments quelconque) de construction d'une instance d'arbre, les instances d'arbre x, y, z. Si on exécute la fonction f avec, successivement, les arguments x, y, z, alors f produira une instance d'arbre avec, respectivement, comme premier, deuxième et troisième fils, x, y et z.

Ainsi, pour inserer le nouveau fils à la position choisie, il suffit de choisir sa position lors du passage d'arguments.

De même on peut facilement déduire la conception de la fonction de suppression.

R E A L I S A T I O N

Notons que la fonction informative ETAT produisant la classe du fils (ou racine) désignée par la position courante peut être utile pour connaître la classe du fils (ou racine). La fonction TYPE, produisant le type du fils (ou racine) désigné par la position courante, peut être utile, notamment pour assurer la précondition

(concordance en type) aux fonctions REMPLACER, SUPPRIMER et INSERER.

Arité fixe

REMPLACER x n y

ENTREE
- x : l'instance d'arbre
- n : numéro du fils au noeud désigné par la position courante (*)
- y : la nouvelle instance d'arbre qui va remplacer ce n fils

SORTIE
- t : le remplacement a été effectué
- nil : sinon

EFFET le n fils de l'instance d'arbre x est remplacé par le nouveau fils y.

PRECONDITION
la concordance en type entre le fils remplacé et l'instance d'arbre remplaçante

COPIER x n

ENTREE
- x : l'instance d'arbre
- n : numéro du fils au noeud désigné par la position courante (*)

SORTIE si la copie a été effectuée, l'instance d'arbre représente le n fils de x;
sinon nil

EFFET fournir une copie physique du n fils de l'instance d'arbre s'il existe

---

(*) celle étant attachée à x et transparente à l'utilisateur.

Arité variable

INSERER x n y

PRECONDITION	le type de l'instance d'arbre y (nouveau fils) concorde avec celui du noeud de la position courante (*)
ENTREE	x : l'instance d'arbre n : entier supérieur ou égal à zéro y : l'instance d'arbre représentant le fils à insérer au noeud indiqué par la position courante (*)
SORTIE	t : l'insertion a été effectuée nil : sinon
EFFET	insérer l'instance d'arbre après le n fils de l'instance d'arbre x; cas particulier : n=0 l'instance d'arbre y est inséré en première position

SUPPRIMER x n

ENTREE	x : l'instance d'arbre n : entier supérieur à zéro
SORTIE	t : la suppression a été effectuée nil : sinon
EFFET	le n fils du noeud indiqué par la position courante (*) est supprimé

(*) celle étant attachée à x et transparente à l'utilisateur.

Terminaux à propriété

LIRE x n

ENTREE x : l'instance d'arbre
n : entier supérieur à zéro

SORTIE la propriété attachée si elle existe
nil : sinon

EFFET la propriété attachée au n fils du noeud indiqué par la position courante (*) est délivrée

ECRIRE x n P

ENTREE x : l'instance d'arbre
n : entier supérieur à zéro
P : la propriété à attacher

SORTIE t : la propriété est attachée
nil : sinon

EFFET la propriété fournie en entrée est attachée au n fils à condition que
- le fils admette une propriété
- la propriété réponde aux règles associées (famille F3, chapitre 3, p 36)

(*) celle étant attachée à x et transparente à l'utilisateur

3.1.6. Famille 6 : LES META-CONSTRUCTIONS (F6)

Rappelons que, pour disposer d'une instance d'arbre relative à une C-G, il suffit d'exécuter la fonction (appartenant à F1 ou à F2) correspondante et que, parmi les fonctions appartenant à F1 ou à F2, les fonctions associées aux C-G à arité fixe requièrent un nombre fixe d'arguments. En pratique, il arrive que l'on ne dispose pas de tous les arguments. Cependant on voudrait une instance d'arbre même "incomplète".

Considérons le cas où on veut construire une instance d'arbre représentant un objet, la fonction construisant les instances d'arbre correspondant aux objets étant d'arité fixe. Chacune des instances d'arbre réserve quatre fils respectivement pour l'identificateur, le type, la définition formelle et la définition informelle de cet objet. Supposons que l'on dispose seulement de l'identificateur et du type de l'objet, on voudrait avoir l'instance d'arbre représentant cet objet où le premier et deuxième fils auront respectivement comme propriété l'identificateur et le type de l'objet tandis que les troisième et quatrième fils restent "vides". Ce cas-ci se produit lors de l'analyse syntaxique. Supposons qu'on reconnaisse l'identificateur et le type d'un objet; on souhaite construire l'instance d'arbre "incomplète" représentant cet objet, la raccrocher à un fils de l'instance d'arbre représentant l'énoncé considéré. Plus tard, on complétera respectivement les troisième et quatrième fils de l'instance d'arbre de l'objet par les instances d'arbre représentant les définitions formelle et informelle

Exécuter une fonction pour disposer d'une instance d'arbre est une solution. Il en existe une deuxième plus rapide qui consiste à stocker en mémoire une instance d'arbre "type" (voir la définition plus loin) pour chaque C-G. Si besoin en est, il suffit ainsi de prendre une copie de l'instance d'arbre stockée en mémoire.

C O N C E P T I O N

Face à ces deux solutions, nous nous trouvons en face du conflit "temps d'exécution - place mémoire". Avant d'opter pour une solution, nous nous attarderons sur les C-G.

Rappelons que les C-G ne sont qu'une autre présentation schématique de la grammaire du langage pour laquelle on peut remarquer qu'un certain nombre de production seront probablement plus utilisées que d'autres. En outre, lors de l'association d'une construction associée (cf. chapitre 2 page 24), on utilisera certaines C-G plus que d'autres. On a ainsi subjectivement déterminé un ensemble des C-G les plus employés. A chacune des instances d'arbre correspondant et appartenant à cet ensemble déterminé des C-G on a stocké une copie en mémoire.

Abordons à présent le problème des arguments. Voici des exemples.

Les arguments d'une fonction (de F1 ou de F2) fournissant une instance d'arbre associée à une C-G sont eux-mêmes des instances d'arbre associées à des C-G et récursivement. On s'arrête lorsqu'on aboutit à des C-G représentées par des instances d'arbre ayant des fils terminaux. Ainsi un argument peut être une composition d'un grand nombre d'instances d'arbre correspondant à plusieurs C-G. Comme il a été indiqué, il arrive que l'on ne dispose pas d'un argument. Dès lors, on a fourni des "jetons". Un "jeton" est une instanc d'arbre "minuscule" ayant aussi un type (univers en CEYX). Sachant que les arguments fournis à une fonction doivent concorder en type (cf. famille F2, page 34), on a fourni un ensemble de "jetons" de différents (*) univers (ou types) à l'utilisateur à qui il appartiendra de remplacer, s'il le souhaite, les "jetons" par les instances d'arbre "réelles".

R E A L I S A T I O N

Les copies de l'ensemble déterminé des instances d'arbre ont été produites par les fonctions de F1 et de F2. Lors du chargement du système Lisp, les copies sont assignées à des variables globales. Pour éviter toute confusion avec les variables-utilisateurs, les identificateurs des variables globales commencent par le caractère "&". Ainsi on a une instance d'arbre "type" pour un objet appelé &objet.

(*) le sous-univers n'étant pas pris en compte.

3.2. NIVEAU 2 : LA VISUALISATION DES INSTANCES D'ARBRE

A l'aide des familles appartenant au niveau 1, nous pouvons construire des instances d'arbre, les parcourir et les manipuler. La forme interne dans laquelle les instances d'arbre sont stockées est peu lisible. Le but de ce niveau est de permettre une visualisation plus agréable.

3.2.1. Famille 7 : LA RECOLTE DES INSTANCES D'ARBRE (F7)

Habituelleemnt, le passage d'une forme interne à une autre forme externe est appelé décodage. En édition syntaxique, ce passage est appelé la décompilation. Elle consiste à présenter l'instance d'arbre sous une forme agréable à lire en ajoutant des espaces d'indentation, des mots-clé, la ponctuation, etc.

On peut décomposer le problème de la visualisation en deux sous-problèmes. Le premier est le parcours des instances d'arbre, le second la présentation. Bien que cette dernière soit importante pour une lecture aisée, elle reste un problème de convention, de choix. Ainsi, on peut décider sans difficultés le nombre de blancs à décaler ou le nombre de lignes à passer. Il nous reste alors le problème de parcours.

Rappelons que nos instances d'arbre sont construites de façon incrémentale (cf. chapitre 1, page 13). Ainsi une instance d'arbre est elle-même une composition d'autres instances d'arbre. Nous savons que chaque instance d'arbre représente une C-G. Dès lors, on peut ramener le problème de parcours d'une instance d'arbre à celui du parcours de chacune des instances d'arbre qui la composent. De façon similaire, on peut ramener le problème de présentation.

C O N C E P T I O N

Les "actions sémantiques" et la fonction SEND de CEYX seront exploitées ici pour résoudre la visualisation des instances d'arbre. La résolution suivra la méthode esquissée.

Rappelons que CEYX permet d'associer des actions dites sémantiques. Ces actions ne sont rien d'autre que des fonctions Lisp qui sont regroupées en une fonction Lisp (une fonction par constructeur).

La fonction SEND x où x est une instance d'arbre a pour effet d'appeler la fonction (les "actions sémantiques") associée au constructeur de chaque noeud de x. Le "parcours" de x est à la charge de l'utilisateur.

Pour résoudre ce problème, on a associé une fonction (*) à chaque constructeur (**) où chacune de ces fonctions a la structure suivante :

```
DEBUT
  . Selon le type de l'instance d'arbre associée à ce
    constructeur :
      ajouter mots-clé, ponctuation et faire indentation
      si nécessaire
    Pour chaque fils de l'instance d'arbre associée à
    ce constructeur, effectuer :
      si le fils est à propriété
        alors afficher la propriété
        sinon appeler la fonction SEND pour chaque fils (***)
              de l'instance d'arbre associée à ce constructeur
      fin.
  fin.
FIN
```

Notons ici que ce processus de visualisation est récursif.

(*) regroupant les actions sémantiques.

(**) cf. définition, chapitre 1, page 19.

(***) l'accès au fils se fait par DEPLACER ou GETSON.

R E A L I S A T I O N

IMPRIMER x

ENTREE x : l'instance d'arbre

SORTIE les spécifications représentées par cette instance d'arbre (*)

EFFET les spécifications représentées par cette instance d'arbre sont présentées suivant la notation infixée

PRECONDITION
x est une instance d'arbre

AFFICHER x

Cf. la fonction IMPRIMER mais la présentation est en notation préfixée

---

(*) La décompilation commence par le fils (ou racine si le chemin est vide) indiqué par la position courante.
Ainsi on peut décompiler, par exemple, seulement, un fils de l'instance d'arbre.

3.2.2. Famille 8 : PARAMETRAGE (F8)

Dans la famille F7, l'instance d'arbre est visualisée à partir du fils (ou racine) indiqué par la position courante jusqu'à l'aboutissement aux fils terminaux. A chaque noeud visité on visualise les fils de celui-ci. Afin de produire un récapitulatif, il est utile de ne pas visualiser tous les fils d'un (ou de plusieurs) noeuds.

C O N C E P T I O N

Il existe plusieurs solutions pour fournir un récapitulatif.

Une première solution consiste à déterminer a priori les récapitulatifs. Cette solution est rejetée pour deux raisons principales : elle est rigide, non évolutive selon les besoins; elle est en contradiction avec la philosophie de développement fixée dès le départ (concevoir des outils souples et non figés).

Une deuxième solution consiste à fournir à l'utilisateur des fonctions lui permettant de choisir afin de répondre à ses propres besoins.

Rappelons le procédé suivi pour la visualisation d'une instance d'arbre : on visualise successivement les instances d'arbre qui la composent. Le processus s'arrête quand on atteint les instances d'arbre à fils terminaux.

Rappelons également que

- chaque instance d'arbre possède un type;
- chaque énoncé est une instance d'arbre constituée de quatre fils réservés respectivement pour l'identificateur, le(s) objet(s) "résultat", "intermédiaire" et "donnée";
- chaque objet est une instance d'arbre constituée de quatre fils réservés respectivement pour l'identificateur, le type, la définition formelle et la définition informelle de l'objet;
- il existe deux types de définition formelle : la définition dite explicite et celle dite implicite.

Le paramétrage consiste, pour l'utilisateur, à décider, par exemple de :

. Ne plus visualiser (*) que certains fils d'une instance d'arbre de tel type;

. Ne plus visualiser (*) un ou plusieurs fils de toute instance d'arbre représentant un énoncé;

. Ne plus visualiser (*) un ou plusieurs fils de toute instance d'arbre représentant un objet;

. ne plus visualiser (*) les définitions formelles explicites (ou implicites) des objets.

Pour la réalisation, nous avons paramétré les fonctions de visualisation de la famille F7. Ce paramétrage est interne et non accessible à l'utilisateur. Nous avons crée deux fonctions permettant de paramétrer la visualisation.

Notons l'utilité de ce paramétrage en citant quelques exemples. Ne plus afficher le 4e fils des instances d'arbre représentant les objets. A chaque invocation des fonctions de F7 (IMPRIMER ou AFFICHER), la définition informelle des objets n'est plus visualisée. De même, on peut décider de ne plus visualiser le troisième fils des instances d'arbre représentant un énoncé. Pour chaque fonction de F7, les objets "intermédiaire" ne sont plus visualisés

R E A L I S A T I O N

INHIBER cons n

ENTITE cons : le nom du constructeur
n : entier supérieur ou égal à zéro

SORTIE t : la décision est acceptée;
nil : sinon

EFFET . n 0 Aux prochains visualisations, le n^{q} fils de chaque instance d'arbre construite par le constructeur "cons" n'est pas visualisé
n=0 Aux prochaines visualisations, chaque instance construite par le constructeur "cons" n'est plus visualisé

(*) La décision est maintenue jusqu'à l'exécution de la fonction inverse de celle qui a réalisé l'exécution (cf. partie Réalisation).

EXCITER cons n

C'est la fonction inverse de INHIBER.
Le profil de EXICITER est le même que pour la fonction précédente sauf qu'elle "inverse" la décision prise avant

3.3. NIVEAU 3 : L'EXPLORATION DES INSTANCES D'ARBRE

Les fonctions appartenant à ce niveau ont pour objectif de fournir un diagnostic (résultat d'un ou plusieurs énoncés) sur la forêt d'instance d'arbre

3.3.1. Famille 9 : AIDE (F9)

Toutes les familles (sauf F7) introduites aux niveaux antérieurs traitent les instances d'arbre sans se préoccuper du fait qu'elles représentent des énoncés en langage SPES. Les fonctions suivantes vont prendre l'aspect "sémantique".

Rappelons qu'un énoncé est une suite de définitions d'objets, chaque objet possédant un identificateur, une sorte, une définition formelle et une définition informelle.

Soit l'instance d'arbre représentant un énoncé quelconque. On peut se poser par exemple, les questions suivantes :

1. Existe-t-il un (des) objet(s) de tel type (ou sorte) ou encore existe-t-il un (des) objet(s) ayant tel identificateur ?
2. Quel(s) est (sont) l' (les) objet(s) participant à la définition formelle de tel objet ?
3. Quel(s) est (sont) l' (les) objet(s) au(x)quel(s) tel objet contribue à sa (leur) définition formelle ?
4. Trier topologiquement le(s) objet(s) d'un énoncé, la relation entre objets étant : x et y étant des objets, s'il existe une relation de x vers y, alors l'objet x utilise l'objet y pour sa définition formelle.? En absence de circuit, le tri nous stratifie les objets ce qui facilitera la lecture et sera aussi un outil important lors de la transformation des énoncés en vue d'un ordonnancement de calcul.

On peut aussi relever d'autres questions relatives à un ensemble d'énoncés.

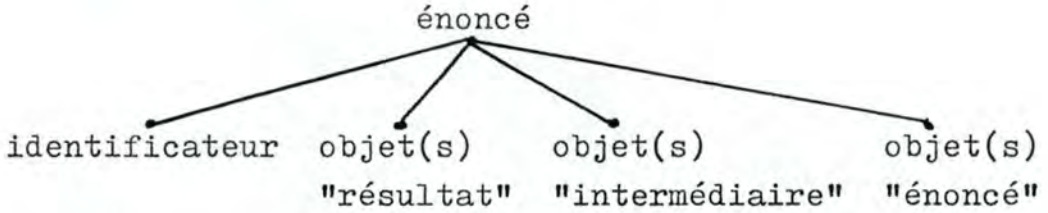
5. Quel(s) est (sont) l'(les) énoncé(s) appelé(s) par tel énoncé ?
6. Quel(s) est (sont) l'(les) énoncé(s) faisant référence à tel énoncé ?
7. Trier topologiquement les énoncés, la relation entre énoncés étant : x et y étant des énoncés, s'il existe une

relation de x vers y, alors l'énoncé x fait référence à l'énoncé y ?

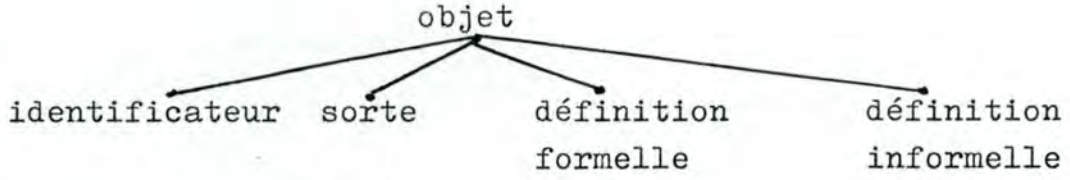
Enfin, pour revenir au concept "jeton" introduit précédemment (F6), on peut s'interroger si telle instance d'arbre (*) contient des "jetons".

C O N C E P T I O N

1. Afin de répondre à la première question, examinons d'abord la structure de l'instance d'arbre représentant un énoncé. Cette structure a déjà été introduite auparavant : l'instance d'arbre d'un énoncé possède quatre fils. Les trois derniers fils sont réservés pour contenir respectivement le(s) objet(s) "résultat", le(s) objet(s) "intermédiaire", le(s) objet(s) "donnée"



Quant à celle d'un objet, sa structure est



Pour déterminer s'il existe un (des) objet(s) de tel identificateur ou de telle sorte, il suffit de se positionner successivement sur les trois branches de l'instance d'arbre d'un énoncé et d'examiner chaque fois si le(s) objet(s) raccroché(s) présente(nt) la propriété (identificateur ou sorte) demandée. De cette façon, on a conçu la fonction SELECTIONNER opérant comme ci-dessus. Cette fonction produit le(s) numéro(s) du(des) fils possédant la propriété (l'identificateur ou le type) spécifiée s'il existe.

(*) où la position est indiquée.

2. Après avoir localisé l'objet (s'il existe) (avec la fonction SELECTIONNER), on pourrait procéder, de façon analogue, pour résoudre cette deuxième question. Mais un examen rapide de la structure d'instance d'arbre engendrée par la définition formelle d'un objet quelconque nous révèle qu'elle peut peut être complexe, ce qui rend plus difficile la tâche d'exploration. Le procédé que nous allons suivre est lié à celui utilisé lors de la décompilation (cf. famille F7, page 60). Lors de la décompilation, les fonctions attachées ont pour mission la présentation et le parcours de fils. Dans ce cas, la tâche des fonctions sera de déterminer la présence d'un objet et le parcours des différents fils. Si un objet a été détecté, son identificateur est stocké pour être restauré plus tard. A la fin du parcours, les identificateurs des objets utilisés (faisant partie de la définition formelle de cet objet) sont renvoyés.

3. Pour cette troisième question, le même procédé sera utilisé que lors de la réponse à la deuxième question. La différence sera d'une part la consultation de tous les objets présents dans l'instance d'arbre, d'autre part, le fait suivant : lors de l'examen, le stockage n'aura lieu que si l'identificateur de l'objet est le même que celui de l'objet examiné.

4. On peut considérer que l'(les) objet(s) présent(s) dans un énoncé est (sont) le(les) sommet(s) d'un graphe orienté. La présence d'un arc partant d'un sommet (ou objet) à un autre signifie que le sommet (ou objet) origine utilise le sommet (ou objet) destinataire pour sa définition formelle.

Remarquons que les réponses aux questions 2 et 3 fournissent respectivement le demi-degré intérieur et extérieur d'un sommet (ou objet). Afin de répondre à cette question (le tri topologique des objets), il faut

. disposer tous les objets (sommets)

. calculer à chaque objet le demi-degré intérieur et extérieur.

Ainsi, on peut dresser ce graphe. Il reste alors à trier. Aucune précondition n'est posée sur le graphe. Celui-ci peut contenir un (ou plusieurs) circuit(s). Dans ce but, on a choisi un algorithme de tri qui n'impose pas de précondition sur le graphe à trier. L'algorithme et la démonstration peuvent se trouver dans [FICH EF].

Les questions 5, 6 et 7 peuvent se résoudre comme les questions 2, 3 et 4 en remarquant qu'ici, l'énoncé est pris en considération.

8. Un examen du (des) fils (ou de la racine) indiqué par la position courante peut facilement résoudre le problème. Ainsi l'utilisateur peut contrôler si le(s) fils (ou la racine) indiqué(s) par la position courante est (sont) complet(s).

R E A L I S A T I O N

1. SELECTIONNER x n p

PRECONDITIONS	x : instance d'arbre représentant un énoncé la position courant indique le noeud des objets "résultat", "intermédiaire" ou "donnée"
ENTREE	x : l'instance d'arbre n : entier supérieur à zéro p : la propriété
SORTIE	liste nil
EFFET	fournit la liste du (des) numéro(s) du (des) fils indiqué par la position courante dans l'instance d'arbre x ayant la propriété p à son (leur) n fils; nil si aucun objet ne satisfait la propriété

2. USER id

ENTREE	id : l'identificateur d'objet
SORTIE	t nil liste
EFFET	sélectionner l'objet d'identificateur id pour produire en sortie . t si l'objet existe et n'utilise aucun objet pour sa définition formelle; . nil si l'objet n'existe pas ou s'il existe un circuit dans le graphe associé à la définition formelle de ces objets;

3. SERVIR id

ENTREE	id : un identificateur d'objet
SORTIE	t nil liste
EFFET	explorer les objets pour produire . t : si l'objet en entrée existe mais ne participe à aucune définition d'un autre objet; . nil : si l'objet en entrée n'existe pas; . liste autrement : liste d'identificateurs où l'objet d'identificateur "id" participe

4. STRATIFIER id

ENTREE	id : un identificateur d'énoncé
SORTIE	liste nil
EFFET	tri topologique du (des) objet(s) de l'énoncé spécifié(s) en entrée. La liste produite est composée des sous-listes représentant dans l'ordre les objets de niveaux 1, 2, ... nil si un circuit existe

5. COMPULSER x

ENTREE	x : l'instance d'arbre
SORTIE	liste nil
EFFET	examiner si les différents fils du noeud (ou de la racine) indiqués par la position courante son des "jetons"; liste des numéros des fils où il y a un "jeton"; t est renvoyé si aucun "jeton" n'a été décelé

3.4. NIVEAU 4 : L'EDITION

Le sujet principal des niveaux précédents est celui des instances d'arbre. Nous allons à présent nous occuper du langage SPES pour mettre en oeuvre un analyseur syntaxique. Le chapitre suivant traitera ce niveau.

CHAPITRE 4 GENERATION DE L'ANALYSEUR SYNTAXIQUE ET REALISATION DE L'ANALYSEUR LEXICAL

A famille 1, nous nous sommes dotés de fonctions produisant des instances d'arbre associées aux C-G. Il a aussi été indiqué que ces fonctions nous permettent de construire l'instance d'arbre associée à un énoncé.

Dans ce chapitre, nous allons procéder à cette construction incrémentale. Dans ce but, nous allons générer un analyseur syntaxique et réaliser un analyseur lexical en langage SPES.

4.1. Famille 10 : TRANSMUTATION (F10)

L'ANALYSEUR SYNTAXIQUE

4.1.1. Génération de l'analyseur syntaxique

Nous disposons du générateur d'analyseur syntaxique LEYACC. Ce générateur demande en entrée un fichier décrivant la grammaire du langage pour lequel on souhaite avoir un analyseur syntaxique. Ce fichier descriptif se compose de trois parties .

Première partie : Les déclarations

Dans cette partie, on déclare les "tokens" (unités lexicales) et la priorité des opérateurs du langage. Pour ces derniers on déclare aussi l'associativité choisie (gauche ou droite).

Deuxième partie : La grammaire

Sous une forme proche de la notation BNF on décrit la grammaire du langage. On peut aussi associer des actions à chaque règle (ou production) de la grammaire. Ces actions sont des fonctions Lisp. Au cours de l'analyse syntaxique la fonction (ou les actions associées) à une règle est appelée si cette règle a été reconnue.

Troisième partie : Les actions

Cette partie est facultative. Si les fonctions (ou les actions associées) ne sont pas des fonctions standard de Lisp ou autres mais existantes, on les décrit dans cette troisième partie.

Une fois ce fichier en entrée fourni, le générateur produit l'analyseur syntaxique. Si la grammaire est ambiguë, un message est signalé. On cherchera alors à lever ces ambiguïtés en corrigeant le fichier descriptif en entrée.

4.1.2. Construction d'une instance d'arbre à un énoncé : introduction

Rappelons d'une part que dans la famille 1 nous nous sommes dotés de fonctions produisant des instances d'arbre correspondant aux C-G et que d'autre part les C-G correspondent aux productions de la grammaire (cf. chapitre 2, page).
Pour obtenir l'instance d'arbre correspondant à un énoncé nous allons procéder par construction progressive au cours de l'analyse syntaxique. Cette construction progressive de l'instance d'arbre se fera grâce à la possibilité d'associer des actions aux règles de la grammaire. Ces actions ne sont rien d'autre que les fonctions de F1. L'idée de cette construction sera développé à la section 4.1.4.

4.1.3. Mise en oeuvre

Afin de générer l'analyseur syntaxique pour le langage SPES on doit fournir le fichier requis par le générateur. Ayant la grammaire du langage SPES on a relevé les tokens et les opérateurs et leurs priorités, ce qui constitue la première partie du fichier requis. A chaque règle de la grammaire nous avons repris la fonction appartenant à F1 à lui associer. Cette fonction de F1 produisant l'instance d'arbre est l'action qui sera appelée par l'analyseur syntaxique. Enfin nous devons compléter notre fichier par la troisième partie comprenant les fonctions (ou actions associées). Etant donné que ces fonctions sont décrites dans F1 et que cette partie est facultative, on pourrait être tenté d'omettre cette partie. En réalité il n'en sera pas ainsi. En effet, notre grammaire étant algébrique, elle ne permet pas de décrire exactement la structure du langage. On retrouve ainsi l'exemple classique que tout identificateur doit être déclaré. Cette contrainte ne peut être décrite par les grammaires algébriques. Notons qu'en théorie les grammaires contextuelles peuvent décrire très exactement les langages. Malheureusement, en pratique, elles sont trop complexes à mettre en oeuvre. Dans notre cas aussi, nous avons ce genre de préoccupations:

Rappelons, avant de citer quelques exemples, que nous avons trois classes d'objets : les objets "résultat", les objets "intermédiaire" et les objets "donnée". Chaque objet possède un identificateur, une définition formelle, une définition informelle et une sorte (ou type). Cette définition d'objet est représentée par une instance d'arbre. L'instance d'arbre représentant un énoncé réserve trois fils. Chacun de ces trois fils est réservé respectivement à ces trois classes. Ainsi au premier fils on raccrochera les instances d'arbre des objets "résultat", au deuxième celles des objets "intermédiaire", au troisième celles des objets "donnée". Nous mentionnerons comme première préoccupation la determination de l'objet pour raccrocher son instance d'arbre au fils alloué à cette classe. Le fait suivant constitue une deuxième préoccupation : lorsqu'on rencontre une définition formelle d'un objet, s'agit-il d'un nouvel objet déjà défini et dans ce cas on change l'ancienne définition par la nouvelle ou s'agit-il d'un nouvel objet ?

Afin de répondre à ces préoccupations nous avons crée des fonctions utilisant notamment les fonctions appartenant à F1, F5 et F9. Ces fonctions constituent la troisième partie du fichier.

4.1.4. Construction d'une instance d'arbre à un énoncé : développement

Un énoncé étant une suite finie de lignes de spécification, l'analyse syntaxique se passe ligne par ligne. Afin d'illustrer l'idée de la construction d'une instance d'arbre à un énoncé, considérons les productions suivantes du langage SPES.

Les mots en caractères majuscules sont les non-terminaux.
Les caractères entre quotes sont le terminaux du langage.
Les mots en caractères majuscules soulignés représentent les tokens.

(P1) ENTETE → ID-MAJ '(' L-I ';' L-I ')'
(P2) L-I → ID-MIN ',' l-I
(P3) DEF-TYPE → L-I ':' ID-MAJ

Commentons ces trois productions.

(P1) : Elle correspond à la production relative à l'en-tête d'un énoncé. Elle signifie qu'un en-tête est composé d'un identificateur en majuscules, d'une parenthèse gauche, d'une liste d'identificateurs, d'un point-virgule, d'une liste d'identificateurs et d'une parenthèse droite.

(P2) : Une liste d'identificateurs est composée d'un identificateur en majuscules, d'une virgule et d'une liste d'identificateurs.

(P3) : Cette ligne correspond à la production pour définir le type (ou sorte) d'un objet. Une définition de type (d'un objet ou de plusieurs objets) est composée d'une liste d'identificateur en minuscules, de deux points et d'un identificateur en majuscules.

Considérons maintenant les deux lignes de spécification suivantes.

(L1) PGCD (r;a,b)
(L2) a,b : ENTIER

Voici succinctement le déroulement de l'analyse lexicale et syntaxique. (L1) est dans la mémoire-tampon (buffer). L'analyseur lexical commence son analyse pour reconnaître les tokens. Supposons qu'il reconnaisse le token ID-MAJ. Il le passe alors à l'analyseur syntaxique. Ce dernier essaye d'analyser quelles sont les productions susceptibles d'être appliquées. Considérons que le résultat d'analyse détermine l'application de P1. L'analyseur lexical reconnaît successivement 'r' ';' 'a' 'b' ')'. Après chaque reconnaissance il passe l'entité trouvée à l'analyseur syntaxique. Ce dernier cherche à continuer l'application de la production P1. La ligne L1 étant entièrement reconnue par l'analyseur syntaxique, celui-ci va appeler la fonction (actions associées) attachée à P1. Cette fonction appellera des fonctions appartenant à F1 qui se chargeront de créer une instance d'arbre avec quatre fils. Le premier fils aura la propriété de l'identificateur "PGCD" et les trois autres fils réservés pour les objets "résultat", "intermédiaire" et "donnée" sont "vides". (L2) est à son tour chargé dans la mémoire-tampon (buffer) puis elle est prise par l'anlyseur lexical et l'analyseur syntaxique. Après un "va et vient" entre l'analyseur syntaxique et l'analyseur lexical, ce dernier applique la production (P3) et la chaîne est

On considère l'instance d'arbre à l'énoncé. Une deuxième solution consiste à user de la meta-construction & énoncé (cf famille F8, chapitre 3, page) où on changera la propriété du premier fils (l'identificateur) de cette meta-construction. C'est cette deuxième solution qui sera retenue vu sa rapidité (cf. F8, chapitre 3, page).

(L2) est à son tour chargé dans la mémoire-tampon (buffer) puis elle est rpise par l'analyseur lexical et l'analyseur syntaxique. Après un "va et vient" entre l'analyseur syntaxique et l'analyseur lexical, ce dernier applique la production (P3) et la chaîne est entièrement reconnue. L'analyseur syntaxique appelle alors la fonction (ou actions associées) attachée à (P3). Cette fonction associée va vérifier si les objets a et b existent dans l'instance d'arbre représentant l'énoncé PGCD. Etant donné qu'ils n'existent pas, cette fonction va appeler les fonctions appartenant à F1 et F2. Ces dernières vont créer une première instance à quatre fils où le premier fils aura la propriété de l'identificateur "a", le deuxième celle de type "ENTIER" enfin le troisième et le quatrième réservés respectivement pour la définition formelle et la définition informelle restent "vides". Une deuxième instance à quatre fils sera aussi créée de la même façon pour l'objet "b". Ici aussi les instances d'arbre des objets peuvent être construites à partir de la meta-construction & énoncé d'objet. Des modifications des propriétés sont à faire. C'est aussi cette deuxième solution qui sera retenue pour les mêmes raisons que celle citées plus haut.

L ' A N A L Y S E U R L E X I C A L

4.1.5. Réalisation de l'analyseur lexical

Rappelons que seul l'analyseur syntaxique est généré. L'analyseur lexical est lui à la charge de l'utilisateur.

4.1.5.1. Introduction

En toute généralité, on peut dire que l'analyse lexicale consiste à reconnaître les expressions régulières à partir de la chaîne de caractères fournie en entrée. Ces expressions régulières reconnues sont fournies à l'analyseur syntaxique. Celui-ci essaye de retrouver la règle correspondante ou de continuer une règle déjà commencée. Si une discordance a été détectée, une erreur est déclenchée. Les systèmes générateurs d'analyseurs lexicaux demandent en entrée la description des expressions régulières du langage.

4.1.5.2. Conception

Une étude complète de l'analyse lexicale peut être trouvée dans l'ouvrage remarquable de GRIES[GRI;72]. Nous nous contenterons d'un bref exposé.

Les buts de l'analyse lexicale

Grouper des entités terminales (symboles du langage-source) pour en faire des entités syntaxiques uniques appelées "token" ou entité.

Une entité peut être une séquence de blancs (réduite ou non à un seul blanc), une constante de type connu, un mot réservé du langage-source, un identificateur du langage-source, ...

Spécification de l'analyseur lexical

Etant doné une chaîne de caractères de longueur finie, l'analyseur lexical doit permettre l'identification des entités terminales. Parmi les entités terminales, on distingue les terminaux du langage tel que les caractères de ponctuation ("," ";" etc) ou les opérateurs classiques ("+" "-" "*" etc) et les unités syntaxiques. Parmi ces dernières, on distingue alors les mots réservés du langage, des identificateurs, des constantes etc.

Ainsi on distingue :

- les terminaux du langage : Dans ce cas l'analyseur lexical doit remettre à l'analyseur syntaxique le terminal du langage;
- les mots réservés : L'analyseur lexical doit aussi remettre

le numéro de ce mot réservé. Le numéro du mot peut s'obtenir en appelant une fonction appropriée appartenant aux fonctions de LEXACC.

- les unités syntaxiques autres que les mots réservés : L'analyseur lexical remettra le contenu (ou valeur) de l'entité (ou token) et son numéro. Le numéro peut s'obtenir de la même façon.

Il existe deux modes de fonctionnement des analyseurs lexicaux.

Ils peuvent être appelés pour analyser une chaîne de caractères complète, identifiant toutes les entités de la chaîne.

Ils peuvent être appelés seulement pour analyser l'entité suivante et la remettre à l'analyseur syntaxique.

Dans notre cas nous retiendrons le deuxième mode.

4.1.5.3. Réalisation

Un ensemble de fonctions a été réalisé pour permettre l'identification des entités. Parmi ces fonctions il existe une fonction particulière qui sera appelée par l'analyseur syntaxique. Cette fonction sans argument remplira la mémoire-tampon (buffer) contenant la chaîne de caractères, l'analysera cas par cas (en appelant les autres fonctions) afin d'identifier les entités. Enfin, elle remettra le contenu et leur numéro (cas des unités syntaxiques). Remarquons que cette fonction joue le rôle d'intermédiaire entre les deux analyseurs (l'analyseur syntaxique et l'analyseur lexical).

4.2. Famille 11 : EDITION (F 11).

Cette famille comprend une seule fonction qui a pour but de mettre en oeuvre les analyseurs syntaxique et lexical afin de permettre l'entrée des spécifications écrites en langage SPES.

ENTRER identificateur
avec profil

ENTREE : identificateur : chaîne de caractères
SORTIE : - t
- nil
EFFET : "identificateur" étant l'identificateur d'un énoncé.
La fonction recherche si cet énoncé existe déjà. S'il n'existe pas, un nouvel énoncé est créé et la fonction boucle pour permettre d'entrer des spécifications ligne par ligne du nouvel énoncé. Si l'énoncé existe, les lignes de spécification entrées remplacent les anciennes. On entend par création d'un énoncé la création de l'instance d'arbre correspondante. Si des erreurs de syntaxe sont détectées dans une ligne de spécification, un message est affiché et la ligne est ignorée. Une nouvelle ligne est demandée. Pour sortir de la boucle on entre le caractère "." à la fin de la session. Si l'analyse n'a révélé aucune erreur, si ce n'est des erreurs de syntaxe, la fonction renvoie "t" et l' (les) instance(s) d'arbre est (sont) créée(s). Dans le cas contraire, "nil" est renvoyé.

"Imagination - C'est cette partie
dominante de l'homme [...].
L'imagination dispose de tout;
elle fait la justice et le bonheur,
qui est le tout du monde."

Pascal,
Pensées, 182

CHAPITRE 5 EXTENSIONS DE LA BOITE A OUTILS

Les fonctions qui ont été réalisées ne peuvent répondre complètement aux objectifs du projet SPES. Au stade actuel, on sait entrer une spécification (ligne par ligne) et l'(les) instance(s) d'arbre se crée(nt) automatiquement. On pourrait visualiser ces spécifications à l'aide des fonctions SPES. Pour parcourir les instances d'arbre on dispose des fonctions de la famille F4. Pour manipuler les instances d'arbre, on dispose des fonctions de la famille F6. Enfin, pour avoir un diagnostic (résultat d'un ou plusieurs énoncés) sur la forêt d'instances d'arbre, les fonctions de la famille F9 ont été créées.

5.1. EXTENSIONS AU NIVEAU DE CHAQUE FAMILLE

Certaines familles sont "complètes" : ainsi la famille de déplacement (F4) ou de celle de manipulation (F6). Nous entendons par complétude le fait que la plupart des fonctions recherchées peuvent être obtenues par composition de fonctions existantes. Il appartient à l'utilisateur de composer les fonctions existantes afin de créer des fonctions répondant au mieux à ses besoins. Par contre, d'autres familles, notamment les familles d'aide (F9) et d'édition (F11) ne sont pas "complètes". D'autres fonctions peuvent alors s'y ajouter.

5.2. EXTENSIONS AU NIVEAU DES FAMILLES

L'analyseur lexical réalisé est malheureusement taillé sur mesure. Si des changements de la grammaire du langage surviennent, des modifications sont à apporter à l'analyseur lexical. Il serait très utile de créer un générateur d'analyseur lexical. Ce générateur fera partie de la famille de transmutation (F10) ou pourra constituer une famille à lui tout seul.

L'édition est, à ce stade de la réalisation, ligne par ligne ce qui est une solution provisoire. On pourrait soit enrichir la famille d'édition (F11) soit créer une nouvelle famille qui prendra en charge l'édition plein écran et sa gestion. On pourra ajouter aussi des familles de transformation. Ces familles prennent alors en charge la transformation d'une spécification (ou énoncé) en un programme exécutable. Ces familles de transformation formeront alors un nouveau niveau. Ce qui a été conçu et réalisé porte seulement sur les énoncés, les types (ou sortes) n'ont pas été pris en considération dès le départ. Rappelons que chaque objet appartient à une sorte (ou type). Ce type peut être soit prédéfini soit à définir dans la bibliothèque des types. Cette définition suit une syntaxe formelle. Afin de concevoir et réaliser les outils de description des types de la bibliothèque, certaines familles peuvent être reprises tandis que d'autres doivent être créées. Puisque nous n'avons pas étudié la bibliothèque, nous ne traiterons pas davantage ces problèmes de types (ou sortes).

"Oui, la bêtise consiste à vouloir conclure. Nous sommes un fil et nous voulons savoir la trame. ... Quel est l'esprit un peu fort qui ait conclu, à commencer par Homère ? Contentons-nous du tableau, c'est aussi bon."

Gustave FLAUBERT,
Correspondance,
à Louis Bouilhet,
4 septembre 1850

CONCLUSIONS

Le présent texte ne constitue pas un manuel d'utilisation des fonctions réalisées. Un manuel est à la disposition des utilisateurs.

Rappelons que l'objectif de ce mémoire est la conception et la réalisation de fonctions de gestion d'instances d'arbre. Certaines fonctions serviront à la mise en place d'un éditeur syntaxique, d'autres s'attachent à fournir un diagnostic (résultat d'un ou plusieurs énoncés) sur la forêt d'instances d'arbre, à la visualisation des instances d'arbre ··· A ce stade de réalisation, onze familles ont été conçues et réalisées. Ces onze familles forment les quatre premiers niveaux du système.

. Famille 1 (F1) : fonctions permettant de construire des instances d'arbre; ces dernières sont la base de la représentation des spécifications (ou énoncés).

. Famille 2 (F2) : fonctions permettant de construire des instances d'arbre avec ou sans propriété. Contrairement à celles de la première famille, les fonctions de cette deuxième famille n'ont pas de pré-condition.

. Famille 3 (F3) : fonctions "filtrant" les propriétés attachées aux fils terminaux. Un identificateur d'objet est un exemple de propriété. Le "filtrage" consiste alors à vérifier la syntaxe de cet indentificateur.

. Famille 4 (F4) : fonctions permettant de parcourir les instances d'arbre et les fonctions d'information sur la forêt d'instances d'arbre. Ainsi on a des fonctions permettant l'accès, dans une instance d'arbre, du père aux fils, du fils au père, d'un fils à son frère (droit ou gauche), la destruction d'une racine d'une instance d'arbre. D'autres fonctions permettent de connaître les différentes racines d'instances d'arbre constituant la forêt ou de connaître le chemin parcouru dans une instance d'arbre, etc.

. Famille 5 (F5) : fonctions permettant de "manipuler" les instances d'arbre. La "manipulation" d'une instance d'arbres comprend l'insertion, la suppression d'un fils, le remplacement d'un fils par un autre, l'attachement d'une propriété à un fils terminal.

Les opérations d'insertion, de suppression et de remplacement sont soumises à des pré-conditions.

. Famille 6 (F6) : ensemble de méta-constructions; ces dernières sont des instances d'arbre "type". Elles serviront comme "jeton" (cf chapitre 3, p. 58).

. Famille 7 (F7) : fonctions de récolte d'instances d'arbre, de présentation sous une forme agréable. A ce stade, on dispose de deux formes de présentation : préfixée et infixée.

. Famille 8 (F8) : fonctions pour paramétrer des fonctions de la famille F7, cette dernière s'occupant de la récolte, présentation et visualisation d'instances d'arbre. Rappelons qu'un énoncé possède trois classes d'objet (résultat, intermédiaire, donnée), qu'un objet possède un identificateur, une définition formelle, une définition informelle et un type (sorte). Ainsi, à la visualisation d'une instance d'arbre représentant un énoncé, on peut ne présenter que les objets d'une ou plusieurs classes particulières, ou ignorer une ou plusieurs classes, ... De même pour les objets on peut visualiser par exemple seulement l'identificateur et la définition formelle, l'identificateur et le type, ...
De cette façon, ces fonctions de paramétrage permettent à l'utilisateur de constituer ses récapitulatifs.

. Famille 9 (F9) : fonctions produisant une "cross-reference" pour un objet ou pour un énoncé, des fonctions pour trier topologiquement le graphe relationnel des objets d'un énoncé, le graphe relationnel des références entre énoncés, etc.

. Famille 10 (F10) : analyseur syntaxique généré et analyseur lexical conçu et réalisé.

. Famille 11 (F11) : fonctions pour mettre en oeuvre l'analyseur syntaxique et l'analyseur lexical pour l'entrée des spécifications (ou énoncés) écrites en langage SPES.

L'approche suivie ici "boîte à outils" nous a paru encourageante vu sa souplesse et l'extension possible. Ces deux arguments peuvent permettre à l'utilisateur de couvrir ses propres besoins.

Si la richesse en primitives et donc en possibilité de couvrir

ses besoins) d'une "boîte à outils" est importante, l'organisation de cette dernière est tout aussi importante. Ainsi une "boîte à outils" riche et non ou mal organisée posera des difficultés lors d'une exploitation profonde. C'est pour cette raison que notre "boîte à outils" a été organisée en familles.
Ces dernières sont décomposées en fonctions. Cette décomposition familles-fonctions facilitera peut-être l'exploitation des outils conçus et réalisés.

L'ensemble des familles étant structuré en niveaux (les fonctions au niveau i+j utilisent les fonctions au niveau i ($i,j>0$)).
Cette démarche incrémentale ("bottom-up" process) nous a paru réaliste pour un sujet de recherche (au sein duquel nous avons conçu notre travail) où les perspectives peuvent fluctuer.

Après l'application de la démarche incrémentale, la superposition des niveaux et la décomposition de familles en fonctions, le système ébauché nous a paru encourageant vu la possibilité d'une compréhension aisée, d'extension, de contraction, de modification et de souplesse.

La principale limite de ce travail est relative à l'analyseur lexical. Outre les difficultés que nous avons eu à le réaliser, il autrait été plus utile de fournir un générateur d'analyseur lexical plutôt que d'en réaliser un sur mesure comme nous l'avons fait. La contrainte du temps était présente pour nous en empêcher; la non documentation de l'analyseur syntaxique nous a rendu plus difficile la compréhension de l'interaction entre l'analyseur syntaxique généré et l'analyseur lexical que nous avons développé. Cela a nécessité un grand temps d'étude.

La réalisation a été développée en Lisp. Ce langage nous était inconnu au début du stage. Les toutes premières fonctions réalisées ne reflètent pas nécessairement la "philosophie" des fonctions Lisp.

REFERENCES BIBLIOGRAPHIQUES.

[BER;75] Berry G. I975 ; Bottom-Up Computation of Recursive Programs, Rapport LABORIA n° I33, IRIA Rocquencourt

[BMP;83] Création d'un environnement de spécification pour le langage SPES. Mémoire présenté par Michel BUYSE et Paul VANHEMELRYCK pour l'obtention du grade de la licence et maitrise en informatique. Juin 83. Namur.

[BOD;8I] Bodart François. Eléments de conception et d'analyse des systemes d'information des organisations. Notes du cours présenté à l'ecole d'eté d'informatique. AFCET , THIES (République du Sénegal), Juillet 8I.

[DEL;82] Delsile N. M. , Menicosy D. E. et N. L. Kerth (I982) Tools for supporting structered analysis. In automated tools for information systems design , H.J. vasserman (Eds) , North Holland I982, II-20.

[DEM;78] De Marcro T. Structured Analysis & System Spécification Yourdon Press Hall I978.

[DEM;80] Demuynck M. Chenut S. et Nerson J.M. (I980) Systeme Zaide d'aide à la spécification. Actes du congrés AFCET I980 , I79-I88.

[DON;79] Donzeau-Gouge V. , Huet G. , Kahn G. et Lang B. (I979) Introduction au système MENTOR et à ses applications Actes des journeés sur la certification du logiciel Géneve I979 , II4-I43.

[DUB;82] Dubois E. , Finance J-P. et Axel Van Lamsweerde Towards a deductive approach to information system specification and design. Kyoto Conference I982.

[FICHEF] Ficheffet J. Cours de théorie des graphes (I° Licence) Institut d'Informatique à NAMUR (Belgique).

[FIN;79] Finance J-P. et Derniame J-C. Types abstraits de donneés: Spécification utilisation et réalisation. Ecole d'eté de l'AFCET I979 MONASTIR Tunisie.

[FIN;79] Finance J-P. Etude de la construction des programmes : Méthodes et langage de spécification et de résolution des problemes. These d'etat Universite de Nancy I I979.

[FIN;83] Finance J-P. , Grandbastien M , Levy N. , Queré A. , Souquieres J. Rapport interne 83-R-079 , CRIN , Nancy.

[FRA;8I] Fraser C. W. Syntax-directed editing of general data-structures. In Proc. ACM SIGPLAN/SIGOA Conf. text manipulation. Portland Ore June I98I ACM , N. Y. 8I p. I7-2I

[GER;80] Germain G. Les nouvelles techniques de spécification du logiciel. Journeé de synthèse Congrés AFCET Informatique I980.

[GRI;72] Gries D. Compiler construction for digital computers. Wiley & sons I972.

[HAINN] Hainnaut J-L. Cours de base des donneés (I° & 2° Licence). Institut d'Informatique à NAMUR Belgique. Notes Polycopieés.

[HAM;76] Hamilton M. & Redlin S. Higher order software -A methodology for defining software. TEEE Transac. on software Eng. , vol SE. 2 , March I976.

[IBM;75] IBM Corp. Installation management HIPO - A aid design and documentation technique. Second edition ,May I975.

[KER;8I] Kerighan B. W. et Marhey J. R. The unix programming environment computer I4 (4) , April I98I ; I2-24

[LAM;82] Van Lamswerde Axel. Les outils d'aide au développement: Un aperçu des tendances actuelles. XV journeés internationales de l'informatique et de l'automatisme. Paris , I6-I8 Juin I982.

[LAMS;W] Van Lamswœrde Axel. Cours de génie logiciel (2° Licence). Notes de cours. Institut d'informatique à NAMUR Belgique.

[LIS;74] Liskov B. H. et Zilles S. N. Programming with abstract data types. In SIGPLAN notices 9,4, I974.

[MAS;8I] Teitlman W. et Masinter L. The interlisp programming environment computer. I4 (4) April I98I , 25-33.

[PAI;79] Pair C. La construction des programmes.RAIRO Informatique. vol. I3,2,I979 pp II3-II8.

[QUE;83] Queré A. SPES notice du langage. Rapport interne 83-R-0I4 , CRIN , Nancy.

[RIN;8I] Rin N. A. An interactive applications developement system and support environment. In automated tools for information design. Schneider H. J. et Wasserman A. I. (Eds) North Holland , I982.

[SCH;8I] Robillard P. N. et R. Plamondon. An interactive tool for descriptive operational and structural documentation. Proc. 23 th IEEE computer society Intl. conf , sep I98I , 29I-295.

[TAR;79] Tardieu H. Nanci et Pascot D. Conception d'un systeme d'information -Construction de la base de donneés-. Edition d'organisation , Paris I979.

[TEI;79] Teichroew D. et Hershey E. A. III (I979). PSL/PSA : A computer aided technique for structured documentation and analysis of information processing systems. IEEE transac. sof. Eng. Vol SE-3 (I) Jan. 77,4I-48.

[TEIT;8I] Teitelbaum T. Reps T. The cornell synthesizer A syntax-directed programming environment.In ACM vol 24 , Number 5 , Sep. 8I pp. 563-573.

[WAS;79] Wassermann A.I. USE : A methodology for the design and developement of interactive information systems. In formal models and practical tools for information systems design Shneider H.J. (Ed.) North Holland I979 3I-50.

[WAS;8I] Wasserman A. I. et al.Revised report on the programming langage PLAIN. ACM SIGPLAN notices I6 (5) ; I98I; 59-80

[WEI;76] Weil R. Experimentation du generateur ATLAS ; IRIA ; service technique mai I976.

[WEL;76] Weil R. et Gleizes J. C. Experimentation du generateur ARIANE IV. IRIA service technique d'informatique janvier 76.

[ZIS;78] Zisman M.D. Use of production systems for modelling asynchronous concurrent process. In pattern inference systems ;Watermann D. et Hayes-Roth F. ; Academic Press I978.